Tricking Hardware into Efficiently Securing Software

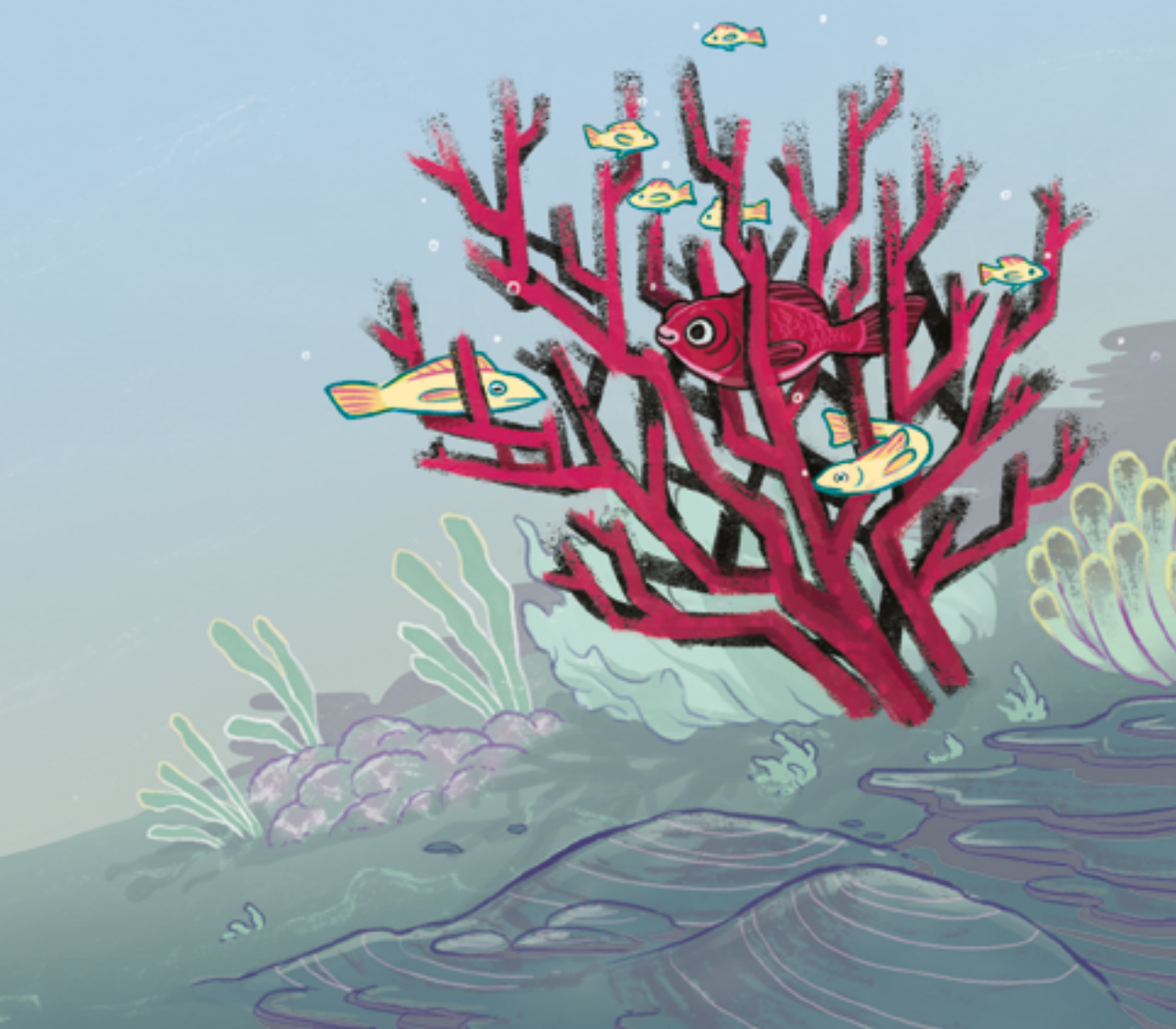
Koning, K.

2021

document version
Publisher's PDF, also known as Version of record

TRICKING HARDWARE INTO EFFICIENTLY SECURING SOFTWARE

Koen Koning



VRIJE UNIVERSITEIT

TRICKING HARDWARE INTO EFFICIENTLY SECURING SOFTWARE

PH.D. THESIS

Koen Koning

The research reported in this dissertation was conducted at the Faculty of Science, at the Department of Computer Science, of the Vrije Universiteit Amsterdam.

This work is part of the research programme *Vici* with project number 639.023.309 titled “Dowsing”, which is funded by the Dutch Research Council (NWO).

Cover: illustration by Mei-Li Nieuwland, typography by Gabor Roozen

VRIJE UNIVERSITEIT

TRICKING HARDWARE INTO EFFICIENTLY SECURING SOFTWARE

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor
aan de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op dinsdag 26 januari 2021 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Koen Koning

geboren te Alkmaar

promotor: prof.dr.ir. H.J. Bos
copromotor: dr. C. Giuffrida

```
/*
 * "Sam sat on the ground and put his head in his hands.  'I wish I had never
 * come here, and I don't want to see no more magic,' he said, and fell silent."
 */
```

`400.perlbench/src/mg.c`, SPEC CPU2006

Acknowledgements

This thesis would not have been possible without the support of family, friends and colleagues. I am grateful to you all for giving me both a great environment to work and learn in, and for helping me maintain my sanity during it all.

Herbert, thank you for being a great supervisor, and for giving me the opportunity to do a PhD in the first place. You taught me a lot about research, and always allowed me to pursue work in areas I was interested in. Moreover, you were always accommodating and supportive when I needed it.[1]

I could not have wished for a better (co)promoter than Cristiano. Your energy and infinite optimism are awesome, and always gave me motivation to persevere. I greatly appreciate how you somehow always seemed to make time for me, be it to help out with a project or to despair together about the state of academia.

I would also like to express my gratitude towards my committee, Pramod Bhatotia, Fabio Massacci, Nele Mentens, Mathias Payer, and Frank Piessens, for taking the time to review this thesis.

During my PhD I had the great pleasure of working together with Taddeüs, both directly on Delta Pointers, and helping me out on other projects (including this thesis!). You taught me most of what I know about compilers, and I am grateful I had someone to share the pain of debugging SPEC with. But moreover, I am happy to have you as a friend.

Alyssa, you are one of the smartest, most helpful and kindest people I know, and you have been supporting me even before joining VUSec. Be it debugging a kernel, fixing a printer, designing a custom GameBoy CPU/PCB, or acquiring cute animal pictures, you were always there to help.

I would also like to thank my former office mate, Kaveh, not only for putting up with this young and naive PhD student back then, but for teaching me so much about academia and research. And of course for the healthy(?) supply of alcoholic beverages.

There are many more (former) VUSec colleagues I would like to thank, for our work together and being there to chat with over lunch or beers. Thank you Andrea,

[1] Just too bad about that whole Emacs thing.

Andrei B, Andrei T, Angelos, Ben, Brian, Chen, Dennis, Elia, Elias, Emanuele, Enes, Enrico, Erik B, Erik vdK, Hany, Istvan, Jakob, Koustubha, Lucian, Manolis, Manuel, Marco, Marius, Michael, Natalie, Pietro, Radhesh, Sanjay, Sebastian, Stephan, Victor D, and Victor vdV. And Caroline and Mojca for shielding me from the dangers of VU bureaucracy,

During my PhD I also had the opportunity to do two amazing internships. Thank you Manuel and others at Microsoft Research Cambridge for teaching me about practical security research, and the wonder of UK pubs. And thank you Sanjay, Philip and the rest of the SAL team at Intel Labs for making me feel part of the team and teaching me so much. Also thank you Dmitrii, Gurunath, Marcela, Palak, and others for the hiking trips and game nights in Oregon.

I count myself lucky to have so many wonderful friends who support me and distract me from work. Thank you Jos for getting me into this computer mess all those years ago, and the beautiful trips to Sweden. Thanks Koen for always looking out for me, and the many evenings of gaming (and sharing this beautiful name of ours). Thank you Arjen, Bas, Cédric, Floris (for so much, including unborking of my Dutch in this thesis), Rex (you were an awesome neighbor), and Victor for the countless evenings and beers we shared the past ten years, may there be many more. Vera, your emotional support has been invaluable, and I can always count on you to cheer me up. Thank you Raphael for giving me my first taste of doing research, and for imparting wisdom on me ever since. Thank you Mei-Li for being able to bring my research to life through illustrations, including the beautiful cover design. Last but not least, for my parents and sister: mijn hele leven lang hebben jullie mij geholpen, in mij geloofd, en voor mij gevochten. Ik kan altijd op jullie rekenen, en zonder jullie was ik hier zeker nooit gekomen. Thank you!

Koen Koning
Amsterdam, The Netherlands, December 2020

Contents

List of Figures

List of Tables

Publications

This dissertation includes several research papers, as appeared in the following conference proceedings. The text differs from the published versions in minor editorial changes made to improve readability:

Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. **No Need to Hide: Protecting Safe Regions on Commodity Hardware.** In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*, page 437. April 23–26, 2017, Belgrade, Serbia.
[Appears in Chapter 2]

Taddeus Kroes[1], Koen Koning[1], Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. **Delta Pointers: Buffer Overflow Checks Without the Checks.** In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*, page 22. April 23–26, 2018, Porto, Portugal.
[Appears in Chapter 3]

Koen Koning, Herbert Bos, and Cristiano Giuffrida. **Secure and Efficient Multi-Variant Execution Using Hardware-Assisted Process Virtualization.** In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '16)*, page 431. June 28 – July 1, 2016, Toulouse, France.
[Appears in Chapter 4]

Sebastian Österlund[1], Koen Koning[1], Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. **kMVX: Detecting Kernel Information Leaks with Multi-variant Execution.** In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, page 559. April 13–17, 2019, Providence, RI, USA.
[Appears in Chapter 5]

[1]Equal contribution shared first authors. Refer to the end of the corresponding chapters for contributions per author.

Related publications not included in the dissertation are listed in the following:

Taddeus Kroes[1], Koen Koning[1], Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. **Fast and Generic Metadata Management with Mid-Fat Pointers.** In *Proceedings of the 10th European Workshop on Systems Security (EuroSec '17).* April 23, 2017, Belgrade, Serbia.

Nathan Schagen, Koen Koning, Herbert Bos, and Cristiano Giuffrida. **Towards Automated Vulnerability Scanning of Network Servers.** In *Proceedings of the 11th European Workshop on Systems Security (EuroSec '18).* April 23, 2018, Porto, Portugal.

[1]Equal contribution shared first authors.

1 Introduction

To err is human. Computer programmers are of course no exception, and computer bugs have existed for as long as computers have. In 1988, such bugs led to the first ever computer worm, the Morris worm, bringing down most of the internet and causing millions of dollars of damage [181]. This incident showed the world that seemingly benign bugs can be abused by attackers, and pose a real security threat. Over the next three decades, computers have gotten increasingly critical to our infrastructure and society, and the financial and social impact of computer bugs is immense [3, 103]. Preventing such errors, however, is *hard*. And while most people agree computer security is important, limited development time and resources often result in flawed security. Ideally, we would instead have the computer itself solve such issues for us, automatically and transparent to the programmer. And while having the computer do such sanity checks on itself is also non-trivial, it could save us a lot of headaches.

The majority of these bugs, including one used over 30 years ago by the aforementioned Morris worm, can be attributed to so called *memory errors* [141, 195, 204]. Such errors can occur in *low-level languages* such as C and C++, where programmers are directly responsible for managing their memory. This error prone process has been eliminated from higher-level languages such as Go, Java, JavaScript, and Python, and in an ideal world all software would be written in such languages. In reality, however, software written in unsafe languages is still ubiquitous for various reasons. For example, languages such as C and C++ provide access to low-level (hardware) features, required by operating system kernels and drivers, and runtimes for higher-level languages. Low-level languages are also often used because they offer higher performance (e.g., browsers, compilers, HPC software). However, in a lot of cases such languages are simply still used for legacy reasons, where migrating an existing codebase would be too costly.

As such, security researchers, software vendors, and hardware vendors have worked extensively on solutions that automatically provide transparent protection [6, 10, 20, 24, 26, 47, 51, 53, 54, 57, 61, 66, 87, 100, 124, 128, 139, 144, 149,

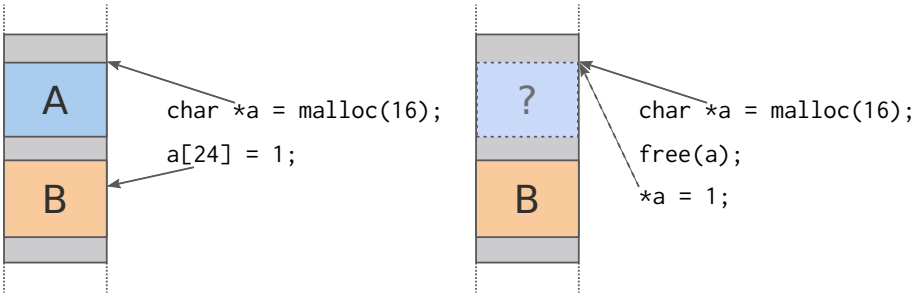
150, 151, 158, 178, 180, 186, 187, 203, 208, 211, 215, 216, 220, 221, 225]. But despite decades of work, these defenses have only limited effectiveness or have not found widespread deployment, and software is still rife with memory errors. The biggest challenges to making memory safety solutions practical are performance overhead and compatibility with existing systems.

Adding memory safety to existing programs generally involves adding additional checks and administration, to ensure the current operation or overall integrity is (still) valid. Doing so naturally incurs some overhead, with more complete (i.e., safer) checks resulting in more overhead [35, 48, 149]. This includes runtime overhead (the program becomes slower) and memory overhead (requires more physical memory to run). Given low-level languages are often used for their high performance and low memory usage, this is a major problem for the adoption of such defenses. As such, a lot of research has gone into optimizing the performance of memory safety solutions, including offering only partial safety for (much) less overhead [23, 66, 138, 203].

Compatibility with existing software is another major hurdle for memory safety solutions [46]. Ideally, memory safety systems would work out-of-the-box on existing software without programmer intervention required. Another facet of compatibility is that of the broader hardware and software ecosystem: solutions that need hardware and operating system modifications require far wider adoption and cooperation from multiple vendors. Ideal solutions simply operate on a single program, and are agnostic of the hardware and wider software on the system. While a structural memory safety design, that is supported by hardware and all software is probably a desirable solution, getting to that point has shown to be difficult so far [8, 48, 116, 148, 160, 214].

Overall, memory safety is far from being a solved problem, with many different aspects and trade-offs. This dissertation on memory safety aims to further our understanding in this space, with new insights and optimizations, to bring us one step closer to eliminating memory errors. This is a difficult area, and this dissertation definitely does not aim to *solve* memory safety once and for all. Instead, this dissertation looks at several issues in this space, and provides new design points.

The focus for this is to improve security, performance, and/or compatibility, in particular by using existing and widely available hardware. New hardware extensions that focus on providing (some form of) memory safety are slowly gaining more traction [8, 94, 158, 160], but developing and deploying them is expensive, takes a lot of time, and does not protect the many existing systems already deployed. We therefore focus solely on what can be done with existing architectures and extensions (e.g., CPU support for virtual machines). This dissertation shows that, by leveraging or repurposing such commodity hardware, we can achieve efficient data isolation, eliminate checks and efficiently monitor running programs. We do so transparently to the programmer, on existing code bases or binaries.



(a) Spatial error **(b)** Temporal error

Figure 1.1: Example of a spatial (a) and temporal (b) memory error. In (a), a buffer `A` of 16 bytes is allocated, but then is written to at offset 24, which will overwrite data of object `B`. In (b), a buffer is allocated, and written to after being freed. The memory may have already been reused in the meantime.

1.1 Memory errors and attacks

Memory errors encompass many of the top 25 most dangerous software errors [142], and constitute over 70% of the security issues encountered by for example Microsoft [141]. In most cases, memory errors are seemingly benign bugs in (low-level) code, and will often simply lead to arbitrary crashes or corruption. However, attackers can use such errors to craft exploits to leak sensitive information, overwrite important data, or gain further access to a system. In this section, we will look at the causes for memory errors, how they can be leveraged for exploits, and what solutions have been proposed.

Memory errors come in two main categories: *spatial* and *temporal*. A *spatial memory error* occurs when the program accesses memory outside of the intended object. The best known example of this is the buffer overflow, like the one used by the Morris worm, as demonstrated in Figure 1.1a. Here the programmer allocated a buffer of 16 elements, but (accidentally or due to attacker-controlled input) accesses the 24th element. This operation is often not detectable, and leads to corruption of other objects in memory. A *temporal memory error* on the other hand occurs when a programmer uses freed or uninitialized memory, as shown in Figure 1.1b. An attacker could massage the memory in such a way that malicious data is placed where the program is inadvertently reading or writing.

While some of these issues are relatively easily mitigated (e.g., uninitialized reads [22, 23, 118, 139]), most memory errors are not. For example, stopping spatial memory error attacks may seem straightforward: simply associate each object with a base and size (or upper and lower bound) and check these before every memory access. While this approach works well for higher-level languages, adding this information to C programs is difficult due to the usage of raw pointers. This

```
void foo(unsigned idx, char **b) {
  char *a = malloc(16);
  a_size = 16;                                                    ①

  if (idx >= a_size) ERROR();                                     ②
  a[idx] = 10;

  b_size = ???;                                                   ③
  if (idx >= b_size) ERROR();
  b[idx] = 12;
}
```

Listing 1.1: Example bounds checks inserted in C code. Each object has an associated size, such as created at ①. For every memory access, the index is checked against this size (②). However, this approach becomes much harder for arbitrary pointers that are passed from other functions or loaded from memory (③).

approach, and its problems, are demonstrated in Listing 1.1. Adding all this information to each pointer, an approach called fat pointers, is known to significantly impact compatibility and performance [10, 99, 151]. Alternative strategies that store this information separately suffer from similar issues [6, 100, 150, 186]. Encoding the bounds information inside the address (i.e., inside the pointer without increasing the pointer size) [64, 113] shows promise, with low memory overheads (e.g., 12%) and moderate runtime overhead (e.g., 64%). We expand upon this idea with Delta Pointers in Chapter 3, and reduce the runtime overhead to 35% by delegating checks to the processor hardware.

Temporal errors are often even harder to solve, with again many performance and compatibility issues. References to an object (i.e., a pointer value) may be copied in the program many times, and approaches that nullify all of these on object deallocation [119, 125, 202, 220] need to keep track of all of these copies. Alternatively, lock-and-key [149, 216] approaches associate a per-object label with each pointer, and check if this label is still valid for every memory access. While only a single label needs to be invalidated on free, keeping track of these references and doing the checks still suffers from compatibility and performance issues. Finally, one could avoid the re-use of memory altogether [54, 62], or replace manual memory management with a garbage collector [4, 26], and while such techniques do better in terms of compatibility, they still suffer from significant runtime and memory overheads. This dissertation does not directly address temporal memory errors, but does shows designs for (partially) mitigating their effects and exploitability in Chapters 4 and 5.

To make matters worse, offering full memory safety requires *both* spatial *and* temporal safety. Many of the aforementioned approaches do not easily compose, with performance far worse than the individual systems, or conflicting altogether.

At the moment no proper full memory safety systems exist, with state-of-the-art solutions incurring a >2x performance overhead and suffering from incompatibility with most common C and C++ programs [35, 48, 149]. Using specialized hardware, such problems can be partially mitigated [48, 148, 187], and we can see a recent interest by hardware vendors to roll out such features [8, 160]. However, such systems are not generally available, and have to make compromises in security to be practical. For example, both SPARC ADI [160] and ARM MTE [8] add identifiers or "colors" to each pointer and chunk of memory, and enforce these colors match on dereference. While this improves performance (e.g., 4-26% overhead [187]), security is limited due to the small number of unique colors (up to 15).

1.1.1 Partial defenses

Both spatial and temporal memory errors allow attackers to leak or corrupt other data in a program, which in turn can be used to mount further attacks, such as corrupt code or data, or hijack the control flow [195]. While memory errors lie at the root of many of these attacks, we saw these are hard to solve. Thus, preventing *further exploitation* instead may be more practical. Many examples of such defenses exist, including stack canaries [50], data execution prevention (DEP/W^X) [137], address-space layout randomization (ASLR) [169], control-flow integrity (CFI) [1], and code-pointer integrity (CPI) [114].

The idea of such approaches is that, in itself, memory errors are not harmful. Exploits often use memory errors to inject code in the program, or overwrite the return address or function pointer. This arbitrary code execution is then used to do further harm. Hence, by protecting return addresses using stack canaries or shadow stacks [50, 114], particular attacks vectors can be mitigated. Similarly, by protecting only all code pointers (e.g., CPI [114]) or limiting where code may jump to (e.g., CFI [1]), more types of exploits can be stopped. Such approaches show that trading in some security can make defenses much more practical. We can see this with DEP/W^X, ASLR, and stack canaries being deployed in virtually all modern systems, despite being relatively easily bypassed. Microsoft has also created a highly optimized (minimal) version of CFI called Control Flow Guard [138], that is enabled for the entire Windows kernel since 2017.

Many of these defenses rely on a separate area of metadata to record information used for checks. Because these defenses are tailored towards preventing certain types of exploits (e.g., ROP attacks) and do not provide full memory safety, this metadata area is not fully protected. An often utilized strategy to protect this metadata area is to simply *hide* it in the (large 47-bit) user virtual address space [14, 20, 43, 57, 114, 144, 222]. Because many attacks have shown this to be an unsafe approach [29, 71, 76, 81, 82, 156], we investigate hardening for such metadata areas in Chapter 2, by leveraging hardware features present in modern processors.

1.1.2 Protecting binaries

Many memory safety solutions can be automatically and transparently applied, but do require recompilation of the program, and thus access to its source code. However, this source code is not always available, for example in the case of proprietary software from vendors, or legacy software where the original code has been lost. As such, defenses that operate on existing binaries are also relevant. Implementing binary-only defenses is often much harder, for example due to the loss of program structure and type information. Binaries can be protected either by changing the underlying hardware, the interface to the rest of the system, or by using (dynamic or static) binary translation, each with its own problems. For example, dynamically instrumenting a binary, where instructions are modified or inserted during runtime, incurs additional overhead. Statically instrumenting a binary, on the other hand, requires complete disassembly of the binary, which is non-trivial to obtain. Additionally, code cannot be easily moved, meaning that inserting instructions requires the insertion of trampolines.

One design that can be applied to binaries is multi-variant execution (MVX), which does not require instrumentation of the binary. Instead of checking the program at a fine-grained level, MVX observes the *behavior* of the program at a high level, such as all I/O operations (i.e., system calls). To detect memory errors at this level, MVX runs multiple variants of the same binary simultaneously, and compares their behavior. The variants are set up such that, during normal operation, they always show the exact same behavior. However, if an attacker tries to exploit the application, at least one of the variants will start showing different behavior.

The security of such MVX systems thus relies on the variant generation being used. The variant generation can be applied at compile time, but also on binaries, for example by mapping the binaries at different locations in the address space, or loading different memory allocators. Such variations in themselves do not perform any checks or provide any security, but merely cause slight divergences in behavior between the variants. As such, they can be lightweight and efficient, and multiple of such variant generation schemes can easily be *composed*. Different parts of the program can be protected against multiple types of memory errors through such composition of variation techniques.

In this dissertation we address several issues on multi-variant execution. MVX suffers from high runtime overheads for monitoring the variants, for which we show an optimized design that utilizes the standard virtualization extensions of the processor. We also increase security of MVX, with more comprehensive variant generation strategies that neutralize attacks relying on memory errors. Both of these are incorporated in our MvArmor prototype, presented in Chapter 4. Finally, we look beyond protecting user-space applications: our kMVX design and prototype, presented in Chapter 5, extends MVX towards operating systems kernels.

1.2 Contributions and roadmap

In this dissertation the goal is to analyze and improve upon the field of memory safety for unsafe languages, in a transparent manner. We do this by leveraging *commodity* hardware features, that can be found already in most available computers. In particular, we address the following challenges and make the following contributions:

In **Chapter 2** we look at existing (partial) memory safety systems and observe many rely on metadata that is merely hidden, but not otherwise protected. This allows attackers to compromise the defense by leaking or guessing the location of this sensitive area. We explore different methods of properly isolating and protecting this metadata, and demonstrate this is practical to do using a variety of different commodity hardware features.

In **Chapter 3** we present our own spatial memory safety system called Delta Pointers, which eliminates separate metadata areas altogether. Moreover, we eliminate the bound checks themselves, by delegating these to the hardware. We leverage pointer tagging and existing hardware (MMU) checks to do so efficiently. By encoding the current distance towards the end of an object in each pointer, we can automatically detect buffer overflows. In addition to this novel bounds checking design, we also provide a study on the impact and compatibility of pointer tagging.

In **Chapter 4** we then explore memory safety for existing binaries (i.e., without recompilation or access to the source code), by looking at multi-variant execution (MVX). This technique runs multiple (diversified) variants of the same program at the same time, and can detect security violations by observing and comparing their behavior. The majority of the overhead for such systems comes from the observing and synchronization of these variants, which is generally done by interposing system calls. By levering existing virtualization hardware extension present in most modern CPUs, we can significantly speed up MVX, as we demonstrate in our prototype called MvArmor.

In **Chapter 5** we extend this MVX design to protect operating system kernels instead. It is often much more difficult to protect kernels than user space programs, and most existing defenses are not applicable to the kernel. With kMVX we present the first prototype for MVX in the kernel, which is aimed at detecting and mitigating information leaks. We show methods of running multiple (modified) Linux kernels on the same system, how to synchronize them, and how to create variance so that information leaks are detected and neutralized.

Finally, we conclude in **Chapter 6** by looking at lessons learned from designing these systems, and look at future directions for memory safety, including recent developments that show promise.

2 No Need to Hide: Protecting Safe Regions on Commodity Hardware

As modern 64-bit x86 processors no longer support the segmentation capabilities of their 32-bit predecessors, most research projects assume that strong in-process memory isolation is no longer an affordable option. Instead of strong, deterministic isolation, new defense systems therefore rely on the probabilistic pseudo-isolation provided by randomization to "hide" sensitive (or safe) regions. However, recent attacks have shown that such protection is insufficient; attackers can leak these safe regions in a variety of ways.

In this chapter, we revisit isolation for x86-64 and argue that hardware features enabling efficient deterministic isolation *do exist*. We first present a comprehensive study on commodity hardware features that can be repurposed to isolate safe regions in the same address space (e.g., Intel MPX and MPK). We then introduce MemSentry, a framework to harden modern defense systems with commodity hardware features instead of information hiding. Our results show that some hardware features are more effective than others in hardening such defenses in each scenario and that features originally conceived for other purposes (e.g., Intel MPX for bounds checking) are surprisingly efficient at isolating safe regions compared to their software equivalent (i.e., SFI).

2.1 Introduction

Operating systems generally provide strong isolation between different processes, but they do not provide any isolation between components *inside* a process. This lack of intra-process isolation allows for fatal memory disclosures in C/C++ binaries, which rank among the most serious vulnerabilities today and have a key role in software exploitation. These memory disclosures allow attackers to reveal the code layout [190] and thus bypass fine-grained diversification [164, 212], leak sensitive data such as cryptographic keys [67], but, most importantly, attackers can bypass state-of-the-art security defenses [114, 129] by deliberately revealing safe regions that host sensitive data of the vulnerable program [71, 76, 81, 156]. In this chapter, we advocate that no matter the defense in place, if deterministic isolation is not guaranteed, then the protection can be bypassed. To achieve such guarantees, we consider efficient isolation techniques based on commodity hardware features. We also introduce MemSentry, a framework that implements these techniques and allows for a comprehensive and practical comparison. Finally, we show that, other than for evaluation purposes, MemSentry can be effectively used to harden modern defense systems that are based on information hiding, including those now deployed in production such as SafeStack [47, 114].

Modern security defenses rely on the confidentiality of their metadata for managing their state. No matter how advanced the protection, any leak of the sensitive metadata is sufficient to subvert it completely. For example, code pointer integrity (CPI) [114] stores all sensitive pointers and related metadata in a so-called safe region which, on 64-bit systems, is hidden at a random location in a very large address space. As a result, the protection of the safe region hinges on the entropy of address space layout randomization (ASLR). Unfortunately, there are *several* different strategies an attacker can follow to bypass ASLR. No matter the size of the virtual address space, attackers can infer the location of the hidden object by leveraging allocation oracles [156], thread spraying [81], crash-resistant primitives [76], or other side channels [29, 71, 82]. All the aforementioned strategies are strong evidence that the probabilistic isolation offered by information hiding should no longer be considered an option as a replacement for deterministic isolation. Instead, hardware features should be revisited for delivering deterministic and efficient memory isolation of safe regions. Where 32-bit x86 architectures provide segmentation to efficiently isolate sensitive data, there is no such answer on modern x86-64 processors. While alternatives to probabilistic isolation exist, they are typically tailor-made for a particular setting. In this chapter we instead propose a general design for memory isolation, allowing any deterministic isolation technique to be applied to the safe regions of any defense. One example is software fault isolation (SFI), which sandboxes code with instrumentation, for instance to ensure attackers cannot access memory outside of a predefined region [219]. While com-

monly applied to sandbox native C/C++ code in the browser, some defenses also adopted it to prevent attackers from bypassing the control-flow integrity (CFI) instrumentation [154]. Recent work on optimizing SFI also supports the importance of efficient isolation in modern defenses, but such work often focusses on a very specific domain. For example, LR^2 [30] reduces SFI checks to protect diversified code only.

The traditional POSIX way of achieving intra-process isolation is by using the `mprotect` system call, which can instruct the kernel to add or remove permissions of memory pages on the fly. Unfortunately, using this strategy to protect safe regions results in significant overhead (e.g., 20–50x in our experiments). We therefore look at hardware features of modern 64-bit Intel processors, including memory protection extensions (MPX), encryption instructions (AES-NI), memory protection keys (MPK), software guard extensions (SGX), and virtualization features such as the new VMFUNC instruction. While researchers have used features like AES-NI [126] or leveraged special x86-64 constructs [59], no prior work has evaluated the pros and cons of the different hardware features for isolating safe regions, nor provided a general design. While we focus on the x86-64 architecture, our approach is also applicable to other architectures when similar hardware features are available (e.g., ARM [192]).

In addition to our analysis of isolation techniques, we also present MemSentry, a comprehensive and self-contained deterministic memory isolation framework. MemSentry serves as a testbed for these techniques, allowing existing and future techniques to be implemented and deployed easily. Not only does it provide a platform for benchmarking and comparing these hardware features, but also to harden modern security defenses that are based on information hiding. To demonstrate its applicability, we evaluate MemSentry against SPEC 2006 and in a variety of scenarios, such as shadow stacks, control-flow integrity, and code diversification. Our evaluation reveals a number of interesting findings, which are currently not well established in the community. We show that MPX—a recently added feature in Intel CPUs for checking pointer bounds—can be repurposed as a replacement to SFI for shadow stacks and other control-flow defenses with acceptable overhead (up to 7.5% vs 21.6% for SFI). Our approach is to rely on a new design where only a single bound check is required, making MPX much more efficient than its normal use case of double bounds checking—rapidly dismissed by practitioners for its exorbitant overhead [158]. On the other hand, we show that for sparse instrumentations such as heap protection, isolation techniques based on features such as VMFUNC are more efficient (5.5%).

To summarize, our contributions are:

- An analysis of recent security defenses, demonstrating their vulnerability against information leakage attacks and stressing the fundamental cause of their weakness: *lack of deterministic isolation for safe regions*,

- A survey of a number of recent (Intel) hardware features and a general design that allows for such features to be used for memory isolation.
- MemSentry, the first comprehensive and self-contained deterministic memory isolation framework, which allows users to benchmark these hardware features on any platform and to strengthen existing defenses for additional protection.
- An evaluation of MemSentry and the various hardware features in different scenarios, ranging from control-flow protection to sensitive user data protection. Depending on the underlying defense, we can protect safe regions with a 1.1% overhead in the best case and with a 2.8% (integrity) or 14.7% (integrity and confidentiality) overhead in the worst case. Our results also provide guidance on how to use different hardware-supported techniques on commodity hardware.

2.2 Memory isolation in review

In this section we give a short introduction of deterministic and probabilistic isolation. Later on, we review some representative defenses that rely on strong isolation for countering exploitation. Finally we discuss the conditions under which probabilistic isolation fails.

2.2.1 Deterministic vs probabilistic isolation

Modern operating systems allow processes to run isolated from each other. Each process has its own virtual address space, and cannot normally interfere with other processes. However, many systems rely on further isolation inside the running process: certain parts of the process (*safe regions*) should be protected from the rest of it.

Isolating certain parts within a running process is commonly referred to as SFI [211]. Depending on the isolation required, implementations may vary, but the basic concept is simple. The isolated part of the process must be compiled masking read and/or write operations, so that all are confined to a given memory range. Any access outside this range, which defines the boundaries of the isolation, is not permitted.

Instrumenting all these operations can be expensive. In 32-bit x86 architectures, where segmentation is available, isolated parts can be placed in different segments [74, 211, 219]. For example, a defense could create a second segment with a higher limit, and place its sensitive data in the high part. Referencing this isolated part can be done only by correctly setting the segment registers, and not by simply overwriting accessible memory. Enforcing isolation by means of SFI, using full segmentation or by instrumenting read/write operations, is what we refer to

Defense	Vuln.		Isolation		Instrumentation points
	r	w	Prob.	Det.	
CCFIR	√		√		Indirect branches
O-CFI	√		√		Indirect branches
Shadow Stack		√	√		`call`/`ret`
StackArmor	√	√	√		`call`/`ret`
TASR	√	√	√		System I/O
Isomeron	√	√	√		Indirect branches
Oxymoron	√		√		Code page across edges
CPI		√	√		Memory accesses
CCFI				√	Memory accesses
ASLR-Guard	√	√	√		Memory accesses
DieHard	√	√	√		`malloc`/`free`
Readactor				√	Indirect branches
LR^2				√	Mem. accesses & ind. branches

Table 2.1: Defense systems that are based on memory isolation. Shown are what vulnerabilities they protect against: reads (r) and/or writes (w), what type of isolation they provide (probabilistic or deterministic) and where they insert code.

as *deterministic isolation*. Deterministic isolation guarantees that the isolated part cannot access anything outside the isolation border. An alternative to deterministic isolation is *probabilistic isolation*, which we can realize using *information hiding*. Rather than truly isolating them, information hiding *hides* the sensitive parts of a running process in a large (64-bit) virtual address space, while removing all references to them from the rest of the process.

2.2.2 Defenses that rely on isolation

Many defense systems for countering software exploitation require a part in memory isolated from the vulnerable process itself. If this part is compromised the defense is rendered useless. Since deterministic isolation is considered expensive in 64-bit systems—for instance because of the lack of segmentation—all of the following systems are realized using information hiding. For better presentation, we have divided the systems in four distinct categories:

Code diversification Code reuse attacks rely on creating a sequence of *gadgets*: small pieces of the original code that an attacker jumps over to create the desired code. For countering such attacks, code diversification destroys any prior knowledge of code locations, so that locating gadgets is no longer deterministic. Diversification can be realized via function/basic-block permutation [212], fine-grained ASLR [78], opcode permutations [164], or instruction-layout randomization [89]. Unfortunately, memory-disclosure vulnerabilities render all these mechanisms ineffective [190]. An additional defense against memory disclosure is run-time rerandomization [42, 78, 213], which first diversifies programs at function or basic-

block level at compilation or at load time, and then periodically remaps the code to different addresses.

For example, Isomeron [57] and Oxymoron [14] consistently maintain offsets which indicate the distance between the currently mapped code and newly remapped code. By adding these offsets to all branch targets, they properly divert the control flow to the newly remapped code. However, if an attacker is able to leak the offsets before remapping is done, they can predict the address of the newly remapped code and bypass the protection. To protect these offsets Isomeron uses a shadow stack-like structure of *recorded decisions*, and Oxymoron uses an indirection table called the *Rattle table*. Isolation of these components is crucial. In a similar fashion, TASR [24] maintains a list of activated code pointers. When remapping occurs, the system remaps these code pointers to point to the newly mapped code. Again, isolation of the list of code pointers is essential, since the attacker could first leak the list of code pointers and then replace them to bypass the remapping entirely.

Control-flow integrity Unlike code diversification, which hides the code space from attackers, control-flow integrity [1] provides more deterministic protection via explicit checking of indirect-branch targets at runtime. CFI instruments every indirect branch to ensure that it can reach only the intended target set as approximated through static analysis. Depending on the number of targets sets, we can classify CFI as either coarse-grained or fine-grained. Coarse-grained CFI [153, 219, 224] supports no more than two or three (large) target sets, making it vulnerable to attacks that construct malicious payloads out of legitimate targets [37, 58, 80]. Fine-grained CFI [56, 70, 205], on the other hand, supports more target sets but also typically introduces more overhead.

Researchers have combined code randomization with coarse-grained CFI to strengthen it, but doing so also introduces an additional component, which must be protected. For example, CCFIR [222] generates code stubs for indirect branches, which it places randomly in its springboard regions, while O-CFI [144] employs a so-called BLT table. In both cases, isolation of these structures is essential.

Code-pointer separation An attacker can hijack the control flow of a program by corrupting a code pointer. One way to prevent such attacks is to store all sensitive code pointers in a well-isolated region of memory. For example, return addresses stored on the stack are high-value targets, and researchers have proposed shadow stacks to protect them from corruption [55, 70, 114]. By storing all return addresses in the isolated "shadow stack", separated from the regular stack, simple stack smashing attacks are no longer possible.

To overcome the problem of leaking the location of the shadow stack, StackArmor [43] allocates individual stack frames at each function call at random

addresses to keep the information leakage attacks from expanding to other stack frames. Alternatively, code-pointer integrity (CPI) [114] introduces a safe region to store both the code pointers and their related metadata, critical to its security.

Another approach is to protect pointers using encryption. Following the idea of PointerGuard [51], which applies a `xor` operation on all pointers with secret keys to prevent buffer overflow attacks, CCFI [132] relies on Intel AES-NI instructions to encrypt/decrypt code pointers. The AES keys are stored in dedicated `xmm` registers. Since CCFI introduces a large overhead (3.5x for SPEC 2006), ASLR-Guard [129] proposes a more lightweight encryption scheme which relies on `xor` instructions. A preallocated key table (AG-RandMap) stores different `xor` keys for each entry, making lookups via code pointers efficient. Moreover, by using different `xor` keys, ASLR-Guard provides stronger encryption compared to PointerGuard, with less overhead. Similar to many previous protections, it is essential to isolate the AG-RandMap not just against information disclosures, but also against writes.

Sensitive non-control data Besides the danger of control data attacks, non-control data can also be security sensitive. Examples include sensitive configuration data, user data, and so on [41]. Such data usually resides on the heap and is therefore vulnerable to information leakage and memory corruption attacks. DieHard [20], and follow-ups [155], address this problem by designing a (probabilistically) safe memory allocator, resilient to heap-based memory corruption.

2.2.3 Threat model

We assume a defense system (see Section 2.2.2) protecting a vulnerable process against code reuse. The attacker holds an arbitrary read and write primitive, but code reuse is not effective due to the defense system in place. Assuming the safe region is *hidden* and not *isolated*, the attacker can carry out the attack in two phases. First, a safe region, important for the defense (e.g., the safe region of CPI [114]), must be revealed using one of the many available techniques [71, 76, 81, 93, 156]. Once the attacker knows the safe region, the defense can be bypassed, and, at this second phase, code reuse can be effectively launched. MemSentry essentially stops the attack at the first phase, protecting the defense system in place, and consequently the protected program. Note that at this point, code reuse is still not possible and MemSentry cannot be attacked by exploiting vulnerabilities and chaining ROP gadgets: executing any ROP gadget can be done only once the defense is bypassed. Finally, we assume that the code of MemSentry is trusted and implemented correctly.

2.3 Deterministic memory isolation

From Section 2.2, it should be clear that memory isolation is critical for many defenses and that their reliance on information hiding is wholly insufficient. In this section, we outline a design for easily-applicable *deterministic* memory isolation. This design can be applied to many existing defenses and abstracts away the details of the underlying features being used. Thanks to our generic design—instead of the ad-hoc solutions of the past—users can now easily swap out different isolation techniques, for instance depending on availability of such features on commodity processors.

We categorize deterministic memory isolation solutions in two groups: *address-based* and *domain-based*. In the former, the address space is split into partitions and memory accesses are masked to only access the allowed partition. Domain-based isolation does not explicitly force memory accesses to go to a certain partition. It instead defines areas in the address space that are completely inaccessible and can be toggled on or off, making them only temporarily accessible to any instruction in the program. One can apply either of these memory isolation categories to a defense system given three sources of information: (a) the safe region(s) of sensitive data (*isolated data*), (b) the instructions of the program that are allowed to access sensitive data (*instrumentation points*), and (c) the preferred memory *isolation technique*.

The remainder of this section discusses this design, starting with the three inputs required, followed by an overview of our deterministic isolation techniques based on commodity hardware features. Figure 2.1 shows an overview of MemSentry, which implements our design using LLVM, allowing it to easily be applied on top of different systems.

Isolated data In general, the sensitive data of a program should be deterministically isolated. Such data may contain cryptographic keys, cookies, access-control metadata, and so on. Additionally, defense systems often contain metadata used internally, which must remain protected. For example, a system may incorporate a safe region containing sensitive code pointers that should not be overwritten or a redirection table which should not be read by other parts of the process.

The effort required to locate the isolated data depends on the particular application. For some defense systems, this is trivial; the memory range occupied by a shadow stack, for example, can be easily determined and isolated. On the other hand, a memory allocator may store sensitive metadata in several places. In that case, modifying the allocator requires additional effort. Systems that are already based on *probabilistic isolation* can always be enhanced to use deterministic isolation, since they already contain a well-defined part that needs to be hidden from the rest of the process.

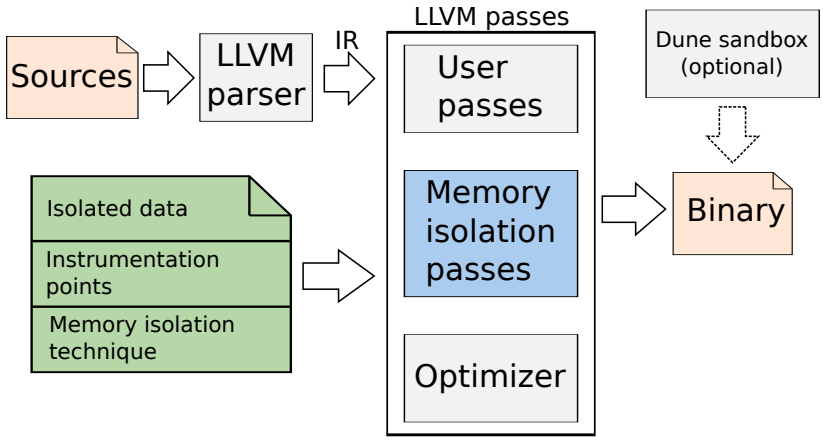


Figure 2.1: Overview of MemSentry's framework. As an LLVM pass it will transform the IR, depending on the user-provided parameters, resulting in an instrumented binary. The resulting binary can optionally be executed in the Dune sandbox to allow for process-level virtualization (e.g., for VMFUNC).

Instrumentation points We refer to the instructions or code regions that need access to sensitive information as *instrumentation points*, as every access has to be instrumented to be permitted. Depending on the application, locating and annotating the code that operates on sensitive data can be trivial or more challenging. For instance, for a typical shadow stack, only the `call` and `ret` instructions are allowed to read and write the sensitive data—in this case an isolated stack containing the return addresses. As another example, consider secure heap allocators such as DieHard [20]. Here the instrumentation points are all calls to the allocator, such as *malloc* and *free*. In both of these cases, locating and annotating the instrumentation points is trivial. However, more problematic cases exist. For example, CPI [114] instruments all read and write operations to code pointers. Inferring all these operations requires points-to analysis and is subject to issues with the accuracy of the analysis. Even in this case, however, determining the set of instrumentation points is already a requirement for the defense itself.

Isolation techniques Finally, one of the many isolation techniques must be chosen to guarantee full safe region isolation. This can be realized using different hardware features available on x86-64. The best choice for the isolation technique depends on the nature of the application. In the following sections, we first detail each of the supported isolation techniques and then discuss the trade-offs among different selections (Section 2.6.3).

Usage From the perspective of a defense developer, MemSentry is easy to use. The developer includes the MemSentry pass to run after the defense pass at compilation time, and adds a static library during linking. The developer then allocates the safe regions using `saferegion_alloc(sz)`, which is part of the static library. For the general case where defense passes insert calls to functions at certain points, these functions should be annotated so they can access the safe region. For the common case where these are contained in a static library, we have included a pass to automatically create these annotations. For more general instrumentation, defense passes can use the function `saferegion_access(ins)` for every instruction that needs access to the safe region. Such an instruction can be a memory access, or for example a call to an intrinsic. The MemSentry pass will run afterward and use the annotations, implemented as LLVM metadata, to insert the correct isolation.

2.3.1 Domain-based isolation

Domain-based solutions split the process operation into multiple *domains*, where a domain can either be *active* (accessible) or *inactive* (triggering faults when accessing data specific to this domain). Domains can contain multiple (smaller) portions of the address space, and are activated using special instructions (which can thus not be triggered by an attacker only equipped with a read/write primitive). For simplicity, we assume a model of two domains: the (default), *nonsensitive domain*, and the *sensitive domain* containing only the sensitive data. Whereas the former is always active, the latter is only enabled when the program accesses isolated data. This model makes it easier to reason about and compare different implementations, but can be extended into multiple and/or disjoint domains, depending on the technique.

For 32-bit x86, segmentation would fall in this category. Sadly, this does not work in x86-64 as segmentation has effectively been removed. Although the `fs` and `gs` selectors still exist, they must point to a location in the accessible address space. We also do not discuss isolation based on traditional paging (optionally sped up using the PCID feature) as this would require intrusive changes to the kernel itself; we aim for a widely deployable and easy to use framework.

Note that, as discussed in Section 2.2.3, we assume the memory isolation works in conjunction with existing defense solutions. Therefore, when protecting such systems we do not require additional domain-switching logic. However, if arbitrary program data should be protected, more complex domain-switching is required [126, 189].

EPT switching via VM functions ("VMFUNC") Intel CPUs include hardware-accelerated virtualization support, called VT-x (or VMX). With hardware virtualization, the host can run several guests efficiently, without emulating them.

Two features that can speed up switching between VMs are the virtual-processor identifier (VPID) and extended page tables (EPT). The former adds tagging information for the TLB, whereas the latter adds a second paging structure. This EPT maps guest-physical to host-physical addresses, that is, physical addresses obtained in the guest via normal page tables are then looked up in the EPT to determine the real physical address. Normally, switching page tables (both normal and extended ones) is a costly operation that only the kernel and hypervisor (respectively) can invoke, making swapping pages in and out impractical. However, with the introduction of VM functions, the guest can invoke several specialized functions (pre-programmed in the CPU) which previously required intervention of the hypervisor. The only VM function currently present is *EPT pointer switching*, which allows the guest to efficiently switch its entire EPT. The hypervisor sets up a list of EPTs, and the guest can then switch between these on its own, selecting the active EPT.

To use these features for memory isolation, we maintain two EPTs, both containing all normal (i.e., non-sensitive) page mappings. However, the mappings for the sensitive data are only present in the second EPT, which is only active when the guest requires access to this data. Thus, we can achieve memory isolation by switching between the normal and sensitive EPTs, representing the *nonsensitive* and *sensitive domains* respectively. By inserting `vmfunc` calls around the instrumentation points, the pages mapped in the secure EPT can be used only by authorized instructions, and thus forms the *sensitive domain*.

Unlike prior `vmfunc`-based solutions, this solution can also function in a self-contained and easily deployable manner by using process-level virtualization [18], as is used in MemSentry. Here, the hypervisor only manages a small VM with the corresponding process running in it, which avoids the overhead and complexity of running the entire operating system and program in a VM. This is also less intrusive than modifying the hypervisor. This aspect is, however, not fundamental to our design; one could also modify an existing hypervisor (such as KVM) to support this.

MPK Memory protection keys will be a feature available in future Intel processors.[1] With MPK, every page belongs to one of 16 domains, determined by 4 bits in every page-table entry (referred to as the *protection key*). For every domain, there are two bits in a special register, named `pkru`. These bits denote whether pages associated with that key can be read or written. While only the kernel can change the key of a page, reading and writing the `pkru` register is possible from user-space using the `rdpkru` and `wrpkru` instructions respectively.

Isolation can be enabled using MPK by placing the sensitive data in pages that

[1] At the moment of writing, Intel has published a full specification [96] on MPK but has not yet announced a processor supporting it.

have a particular protection key, forming the *sensitive domain*. An appropriate instrumentation enables reads and/or writes to the data by setting the *access-disable* and *write-disable* bits, respectively, using `wrpkru`. As long as these bits are unset, the *sensitive domain* is accessible. By setting the bits back, the sensitive domain is disabled, making only the *nonsensitive domain* available.

AES-NI encryption ("crypt") Where the previously discussed domain-based isolation techniques rely on unmapping the sensitive data when it should not be accessed, an alternative is to instead *encrypt* it in-place until it is needed. In order to speed up AES encryption, Intel has introduced dedicated instructions, called AES-NI. This instruction set accelerates the basic blocks of AES: there are functions for performing a single round of encryption and for generating the round keys.

SGX The recently introduced Intel SGX extension can also provide domain-based isolation, even though it is intended for more far-reaching isolation. SGX allows applications to create *enclaves*, separated compartments of code and data. The processor automatically encrypts these enclave compartments, preventing the application itself and even the operating system and hypervisor from reading the enclave's data. This allows for trusted execution in potentially hostile environments, such as cloud infrastructure; clients can upload an encrypted program to a cloud provider, and be guaranteed that it ran correctly. When enclaves are created, the OS sets up the binary blob that should be loaded and the mappings of the enclave, after which it finalizes the enclave. Once an enclave is finalized the application can perform calls into the code of the enclave (ECALLs) via pre-defined entry points. Similarly, the enclave can perform calls outside to the enclave (OCALL), for instance to perform I/O. It is important to note that currently the mappings of the enclave are fixed: no new memory can be allocated to the enclave.

We can apply our domain-based isolation design using SGX by creating an enclave containing the sensitive data and all the code required to access/modify that data. We can thus switch domains by switching execution to the enclave via an ECALL, assuming all code that touches sensitive information can be extracted and combined into the enclave.

However, this approach is currently markedly inferior to the other isolation solutions which we present in this chapter. First and foremost, the overhead is much larger than other solutions, as shown in Table 2.4. Decomposing the application into the domains (the untrusted code and the enclave) is also far less practical: the actual code touching sensitive data must be present in the enclave, whereas with other solutions we simply trigger the domain switch. All sensitive data must be allocated when the enclave is created, and is limited in size, adding more complications in many cases. Finally, deploying this on-the-fly would be more challenging as an Intel-issued signing key has to be used on the binary for the enclave, and

SGX is not yet widely supported by hardware. Processor-level compartmentalization is definitely a promising development, but SGX itself is unsuitable for efficient memory isolation.

2.3.2 Address-based isolation

Address-based solutions split the address space into two or more partitions, instrumenting the program to allow only certain instructions to access a particular partition. This means that every load and store instruction is instrumented with the partition(s) it is allowed to access. For simplicity, we again assume two partitions (the *sensitive partition* and the *nonsensitive partition*), similar to domain-based isolation.

The difference between *address-based* and *domain-based* isolation is that for address-based techniques all pointers are forced into a partition all the time, whereas for domain-based techniques a domain is activated, giving all loads and stores access to it. Additionally, partitions split the address space at a certain point (e.g., everything above 64 TB is sensitive) whereas domains are more flexible.

Traditional SFI ("SFI") One straightforward way of achieving address-based isolation is by using the classic SFI. This can be done at the bit level using software-only instrumentation, requiring no additional hardware support. For example, by storing all sensitive data in the upper half of the address space (forming the *sensitive partition*), the higher bits in a pointer are only set when accessing this sensitive data. By masking the instruction before every non-allowed load and store (using a simple `and` instruction), we can deterministically ensure such instructions cannot access sensitive data. While this approach is widely deployable, it may not be the most efficient solution in many applications of interest (as shown in our evaluation in Section 2.6). Another concern is that traditional SFI cannot deterministically detect invalid memory accesses, but only prevent them, as the masked pointer might still be a valid (different) pointer in the *nonsensitive partition*.

MPX Intel introduced a new hardware feature that allows for efficient bounds checking, called MPX, in the Skylake CPU series. With MPX, the programmer can create and enforce *bounds*, specified by two 64-bit addresses specifying the beginning and the end of a range. Furthermore, new instructions are introduced to efficiently compare a given value against the bounds, raising an exception when the value does not fall within the permitted range. For efficiency, four bounds can be stored into dedicated registers (`bnd0` to `bnd3`). When more bounds are required, they are stored in memory, and the bound registers serve as a caching mechanism. It should be noted that while the bounds-checking itself is very efficient, the usage of many bounds is not. For instance, GCC's implementation of MPX buffer

Isolation	Instrumentation points	Application
Address-based	Loads	Code randomization
	Loads	CFI variants
	Stores	ShadowStack
	Stores	CPI
	Both + points-to info	Program data
Domain-based	`call` + `ret`	ShadowStack
	Indirect branches	CFI variants
	Indirect branches	Layout randomization
	System calls	Layout randomization
	Allocator calls	Heap
	Points-to info	Program data

Table 2.2: MemSentry applies to a range of different defense systems. Some have different variants (e.g., CFI) and the specifics may differ. Nevertheless, the framework is generic and covers various types of applications through different isolation types.

checking frequently spills bounds registers to memory.

For isolation, the address space is partitioned using MPX bounds. By defining a single bound and adding a single bounds check before every memory access, we effectively verify that every pointer used by the program is in the correct partition. By not adding such checks to instructions which are allowed to touch sensitive data, we can enforce that instructions only access memory in the *nonsensitive partition*.

2.4 MemSentry applications

In this section we discuss how our design, and in particular MemSentry, offers deterministic isolation to existing systems, such as the ones presented in Section 2.2, as well as other concerns about the applicability of the framework.

Table 2.2 summarizes the type of isolation and instrumentation points required for some representative systems. For address-based solutions, instrumented memory accesses cannot operate on protected data, while non-instrumented ones can freely read or write to the process address space. For domain-based solutions, the reverse approach is followed. Instrumentation is inserted for accessing sensitive data. For most defense systems, sensitive data is only accessed by new instructions (part of the instrumentation). It is therefore clear which instructions are allowed to touch sensitive data, and no further instrumentation by MemSentry is required. For in-program data such as private keys this is not the case, and MemSentry has to rely on points-to information.

For instance, in a heap protection system such as DieHard, the metadata is only used by the allocator. Therefore, other parts of the program and libraries should not be able to access it. Similarly, access to a shadow stack occurs during `call` and

Isolation technique	Maximum domains	Granularity
SFI	48	–[a]
MPX	4[b]	byte
MPK	16	page
VMFUNC	512	page
AES	Infinite	128 bytes

[a] Depends on least significant bit of mask.
[b] Infinite when also using memory besides bound registers.

Table 2.3: Limitations of memory isolation techniques.

`ret` instructions. Systems based on a shadow stack instrument these instructions therefore it is trivial for MemSentry to determine where domain switches should take place or what memory instructions should not be instrumented for address-based approaches. In this case, only memory stores have to be protected, since the defense is based on the integrity and not the confidentiality of the shadow stack. On the other hand, when protecting private keys, confidentiality is important. Therefore, both reads and writes should be prevented from accessing the cryptographic keys.

Another concern when employing memory isolation is separation of sensitive data from the rest of the program. For many defense systems, separation is enforced by construction. As an example, the code space (a sensitive region in many solutions) is already separated from normal program data in a traditional address space organization. However, arbitrary program data may need protection. In some cases sensitive data is stored in a global data structure, or embedded in another (non-sensitive) one. Table 2.3 shows the theoretical minimum granularity supported by each technique implemented in MemSentry for storing sensitive data. For example, the minimum size of isolated data when using VMFUNC is one memory page. Separation of sensitive and non-sensitive data is carried out by the defense system itself, and MemSentry is employed only for enforcing the desired isolation technique.

As an example, we could apply MemSentry to SafeStack [47, 114], a shadow stack implementation used in production, with minimal effort. This only requires instrumenting all memory writes, and ensuring the normal (safe) stack was located separately in the address space.

2.5 Implementation

In this section we describe some of the details of using the different hardware features in practice, and present our solutions for MemSentry.

2.5.1 VMFUNC

In order to minimize the impact of deploying EPT switching, we deploy this technique (and thus the VM and hypervisor) per process instead of system-wide. To do this, we use a modified version of Dune [18], which allows a single process to run in a VM by running a stripped down version of the KVM hypervisor per process, and a small library-OS to manage the state of the VM. This requires only the (relatively small) Dune kernel module to be loaded, and the remainder of the system does not experience any performance impact.

For MemSentry, we modified Dune to maintain multiple copies of the EPT, which it normally fills in an on-demand basis when an EPT fault occurs. We added a hypercall so that the hypervisor can mark certain mappings as private to only the active EPT, allowing for secret pages to be isolated to a single EPT. The program itself is instrumented to make hypercalls to inform the hypervisor about these secret mappings, and further inserts `vmfunc` calls to switch domains where necessary. The `rax` and `rcx` registers are used to specify the VM function to invoke and which EPTP index to use (respectively). By using the *sandbox* application that is part of Dune, the program runs transparently in the guest environment, with the sandbox taking care of the kernel-level tasks in the VM (such as interrupts and page table management). This technique inherently requires access to certain hardware features such as EPT, VPID and VMFUNC, which might not be implemented in virtual environments (i.e., nested virtualization support is required). However, this is not fundamental to using VMFUNC for isolation, as one could also implement the EPT management in KVM itself.

2.5.2 MPK

As MPK is not yet available in any CPU at the time of writing, we have implemented an approximation that, while not offering any protection, gives performance results and allows for comparison with other techniques.

Normally, an application reads the `pkru` register using `rdpkru`, (un)sets some bits in the result and writes it back using `wrpkru`. Afterwards, memory accesses should use the new permissions as set in `pkru`. To simulate this, we copy an `xmm` register into a general purpose register, set bits in the result, and move it back to the `xmm` register. This approximates the cost of reading and writing `pkru`, as both the `xmm` registers and `pkru` are special registers. In particular, `xmm` registers are the slowest available registers, and so our implementation reasonably approximates the performance overhead of accessing `pkru`. Additionally, we perform an `mfence` instruction to simulate the cost of changing permissions and the probable serialization caused by writing to a control register. MPK itself needs to perform this as well, because memory accesses cannot be reordered around the `wrpkru` calls.

Usage of MPK clobbers `rax`, `rcx` and `rdx` (simulated using inline assembly), pos-

sibly causing compiler register spilling. Since MemSentry runs as a normal LLVM pass, variables are not yet mapped to registers, as this register allocation happens at a later stage. By marking registers as clobbered, LLVM optimizes register usage to minimize spilling to memory. However, both `rcx` and `rdx` are used as function parameters, and are thus often expensive registers to clobber.

2.5.3 Encryption

MemSentry implements encryption using Intel AES-NI with 128-bit keys. This requires 11 rounds for both encryption and decryption, and by extension 11 different round-keys (with one being equal to the overall key). Ideally, none of the keys are ever spilled into memory, as an attacker could potentially *sniff* memory and break the encryption.[2] Furthermore, storing keys in memory is suboptimal performance-wise, as additional memory accesses are introduced.

While pinning certain registers for storing these keys is a solution, as used by CCFI [132], we deemed this approach impractical. First of all, it requires recompilation of all system libraries to mark these registers as reserved, potentially affecting system-wide performance. Secondly, MemSentry requires distinct keys for both encryption and decryption (an encryption round-key can be used to calculate the decryption round-key using `aesimc`), and there are not enough `xmm` registers available to hold these keys. Storing only the primary key (which can be used to generate the round-keys using `aeskeygenassist`) requires fewer `xmm` registers but employs costly keygen instructions for every domain switch.

MemSentry stores the keys in the upper part of the `ymm` registers, which are not used by any of the libraries distributed on Debian and Ubuntu installations. This is more efficient and secure than storing the keys in memory. While this was not necessary for our setup, the compiler can also easily be modified not to use this register with minimal impact [132].

2.5.4 MPX and SFI

Listing 2.1 shows an example of MemSentry's instrumentation. We can see that the calculation of the pointer is separated from the store (the single `mov` becomes a `lea` and a `mov`). For SFI, we load the mask and use the `and` instruction to apply it, using the result for the store. This will force the memory access to always be below 64 TB, although the address space split can be anywhere in the 128 TB address space.

For MPX we insert a single bounds check, which will trigger a fault if the value of `rcx` is above the upper bound of `bnd0`, which we set to 64 TB during program initialization. Because we partition the address space, instead of relying on more fine-grained bounds, only the upper bound has to be checked. The lower bound

[2] We could hide the key in memory using one of the other memory isolation techniques, but this would defeat the purpose of using encryption in the first place.

of the *nonsensitive partition* is 0, and given addresses cannot be negative, a check would be useless. This saves both instrumentation, and slightly increases performance. This implementation does assume the `bnd0` register is not used by the application itself. As MPX is currently only used by compiler passes this is a reasonable assumption, but if an application were to use MPX itself overheads would be larger. Note also that MemSentry enables the `bndpreserve` flag and, therefore, MPX does not reload its bounds (from memory) at any point. Without this flag set, the CPU will load the bounds registers from memory for every branch instruction without a `BND` instruction prefix.

2.5.5 LLVM & points-to analysis

As our instrumentation is performed by an LLVM pass, we can make use of its optimization passes. For instance, in the IR that we instrument LLVM will have eliminated all register spilling to the stack, thus making sure we only see (and instrument) necessary memory accesses. Afterwards, during register allocation in the backend, LLVM might generate variable spills to the stack. These instructions access a fixed place in memory and thus do not need isolation instrumentation.

As shown in Table 2.1, many security defenses instrument solely branch instructions or system calls. MemSentry can trivially infer the instrumentation points in these cases. However, protecting arbitrary information requires further analysis, often called *points-to analysis*, for discovering which instructions operate on which data. In LLVM this can be done statically using the *data-structure analysis* (DSA) pass.

While MemSentry supports DSA, we noticed (similarly to other researchers [203]) that DSA is overly conservative, often yielding undesirable results where most memory accesses are classified as being able to touch sensitive data. We thus also looked at a run-time solution for *dynamic* analysis, in order to approximate an ideal (non-conservative) static analysis only for evaluation purposes. First, the program is prepared with initial instrumentation, which allows later analysis passes to map assembly instructions back to IR instructions. Then, the program is run with a Pin pass [131], which records all accesses to objects per instruction. The output of this run can then be fed back into the instrumentation pass as points-to information. The dynamic analysis itself is much slower, and there is a high chance of under-approximating memory accesses, since only accesses related to particular inputs (i.e., execution paths) are recorded.

We stress here that, even though points-to analysis is an open problem, it does not hinder MemSentry in most cases, where defenses already determine the instrumentation points. In cases where arbitrary data should be protected, any points-to analysis pass can easily be incorporated in the framework.

```
; Store value of rdi in mem
mov     %rdi, 0x8(%rbx)
```

(a) Original

```
; Load pointer into rcx
lea     0x8(%rbx), %rcx

; Faults if rcx is above bnd0
bndcu   %rcx, %bnd0
; Pointer in rcx is now verified
mov     %rdi, (%rcx)
```

(b) MPX

```
; Load pointer into rcx
lea     0x8(%rbx), %rcx
; Mask pointer to be below safe area
movabs $0x00003fffffffffff, %rax
and     %rax, %rcx
; Pointer in rcx is now verified
mov     %rdi, (%rcx)
```

(c) SFI

Listing 2.1: Code transformations caused by our address-based instrumentations. In both cases the calculation of the address and the store have been split (`lea` + `mov`). For MPX, the pointer is checked to be below 64 TB, whereas for SFI it is modified to always be below 64 TB (although this split is arbitrary in both cases).

2.6 Evaluation

In order to evaluate the applicability and trade-offs between the use of different techniques, we evaluate combinations of isolation mechanisms and instrumentation points that are supported by MemSentry on the SPEC CPU2006 benchmark suite. SPEC is very memory and CPU intensive, and thus the overhead for I/O bound applications such as servers will be lower. All benchmarks were performed on a machine with an Intel i7-6700k processor clocked at 4GHz, with 16GB DDR4 RAM. We used Ubuntu 14.04 with Linux 3.19, recompiled to enable MPX support. However, the result for SGX shown in Table 2.4 was performed on the same system equipped with the similar E3-1240v5 processor at 3.5GHz, as not all i7's include

SGX support.

In the rest of the section, we first look at microbenchmarks of all previously discussed techniques, and analyze sources of overhead. Then, we compare the performance overhead of address and domain-based techniques, and show real-world results for SafeStack. Finally, we discuss trade-offs between the two designs, and all the available hardware features.

2.6.1 Microbenchmarks

The results of our microbenchmarks are shown in Table 2.4. We looked at both the cost of memory operations and of our isolation techniques, including some comparable operations. These results serve as a comparison between different operations, and the overall real-world overhead might be lower (e.g., due to pipelining optimizations) or higher (e.g., due to higher TLB pressure). Results were gathered by timing a tight loop of many iterations with the instruction, disabling interrupts and other external factors wherever possible. Our results are consistent with other sources [73, 95].

Memory operations are obviously highly dependant on caching behavior, with spatially and temporally local accesses being far more efficient. In practice the overall observed latency might be lower, depending on the data dependencies. These results give an idea of the cost of spilling registers, which might add to the overhead of some techniques.

The cost of an `and` operation is highly dependent on what is done with the result. When the result is not used, or used as the destination of a memory write, its overhead is not observable, most likely due to effects of pipelining and out-of-order execution. However, when we use the result of the `and` as a pointer for a memory load, its overhead does become significant as this introduces a data dependency.

For MPX we observed a huge performance difference between using a single bounds check, and performing a full bounds check (i.e., checking both the upper and lower bounds using `bndcu` and `bndcl`). The performance difference is mainly because the second bounds check instruction is delayed until the first one completes, whereas the first one does not delay anything. Further experiments confirm this, e.g., executing three bounds check instructions doubles the overhead to $\sim$1 cycle, compared to executing only two. Our measurements thus show that in MemSentry (with a single bounds check and 1 domain), MPX should be faster than SFI in basically all cases, as the overhead of SFI might be observed (when used for loads) whereas the MPX overhead will stay consistently low. In cases where more domains would be required (like GCC's bounds checking) this would no longer be true.

Our virtualization experiments, testing `vmfunc` and a hypercall (`vmcall`), show that the use of `vmfunc` is indeed much more efficient than the old way of involving

Instruction/operation	Cycles
L1 cache access	4[a]
L2 cache access	12[a]
L3 cache access	44[a]
DRAM access	251
SFI (`and`, result used by load)	0.22
SFI (`and`, result used by store)	0
MPX (single `bndcu`)	< 0.1
MPX (both `bndcl` and `bndcu`)	0.50
MPK (simulated)	0.42
`vmfunc` (EPT switch)	147
`vmcall`	613
`syscall`	108
SGX enter + exit enclave[b]	7664
AES encryption and decryption (11 rounds)	41
AES keygen (10 rounds)	121
AES imc (9 rounds)	71
Loading `ymm` into `xmm` (11 times)	10

[a] From Intel [95], in practice these will vary due to access patterns, out-of-order execution and pipelining.
[b] Time of performing an empty ECALL using the Intel SGX SDK for Linux.

Table 2.4: Microbenchmarks for the latency of hardware protection features and related operations.

the hypervisor. We show that the cost of a `vmfunc` is similar to that of a traditional `syscall`.

Our AES results show that round-key generation is far more expensive than fetching the round-keys from the `ymm` registers. It also shows that calculating all required keys for decryption (using `aesimc` 9 times) is far more costly than extracting the normal round-keys (used directly for encryption). While the cost of encryption and decryption per chunk is the same, the initialization cost per block will thus be higher for decryption.

2.6.2 Real-world performance

For a more practical analysis, we run MemSentry on SPEC CPU2006. We present numbers for each benchmark individually, and the geometric mean (geomean) over the set of all C and C++ benchmarks.

As address-based techniques (SFI and MPX) must check the pointers for every memory access, the instrumentation points are different than those of domain-based techniques. In order for address-based techniques to work, *all* memory accesses in an application must be instrumented, with the minor exception of those that are allowed access to sensitive data. In Figure 2.2 we show the results for instrumenting all read instructions, write instructions, and both reads and writes. As we instrument all memory accesses indiscriminately, these figures represent

the worst-case scenario for these techniques. However, as an application mostly deals with non-sensitive data, there is in most cases no difference between instrumenting all instructions, or all minus a few.

As discussed in Section 2.4, the results for instrumenting only reads (*MPX-r* and *SFI-r*, geomean 12% and 17.1%) represent the overhead for CFI and code randomization defense solutions, whereas those with only writes (*MPX-w* and *SFI-w*, geomean 2.8% and 4%) indicate the overhead for protecting a shadow stack. The results for both reads and writes (*MPX-rw* and *SFI-rw*, geomean 14.7% and 19.6%) present a broader (and worst-case) protection, and are applicable when protecting arbitrary program data from disclosure and tampering, such as private keys.

We can see that in almost all cases, MPX performs better than SFI. Both instrumentations take very little time to execute, as shown in Table 2.4. However, in the case of SFI the instruction accessing memory has a dependency on the result of the `and`, whereas for MPX the `bndcl` does not modify the address. While for memory writes this overhead is not noticeable, it causes delays for reads.

Figures 2.3, 2.4, and 2.5 present the results for domain-based techniques. For VMFUNC we switch to a secondary, yet identical, EPT. For crypt, we use AES-NI on a single chunk (16 bytes), and retrieve all round keys from the upper parts of `ymm` registers. Figure 2.3, which shows the results when domain switches occur for every `call` and `ret` instruction, represents the worst-case scenario of all of these, and shows the case of protecting a shadow stack. Figure 2.4 presents a subset of the previous results, where only indirect branches are instrumented, corresponding to CFI and layout randomization solutions. Finally, Figure 2.5 shows results for system calls. We observed similar results for calls to the allocator.

Consistent with the microbenchmarks, MPK is the most efficient of the three with a geomean of 130%, 34%, and 1.1% for `call-ret`, indirect calls and system calls respectively. The costs of crypt (with geomeans of 217%, 60%, and 22%) are higher than might be expected, as not only must the data be encrypted (in 128-bit chunks) but the keys must also be copied into `xmm` registers before encryption can happen. The costs for VMFUNC are generally the highest (with geomeans of 357%, 82%, and 5.5%). Part of this overhead comes from the process-level virtualization, where all system calls are converted into hypercalls. This is especially noticeable for `syscall`-heavy benchmarks, and not as much on SPEC [18]. For benchmarks that already heavily rely on the `xmm` registers, crypt incurs a more significant performance overhead. This is especially evident for several floating-point benchmarks in Figure 2.5, where the encryption itself does not take much time, but clobbering a number of `xmm` registers does.

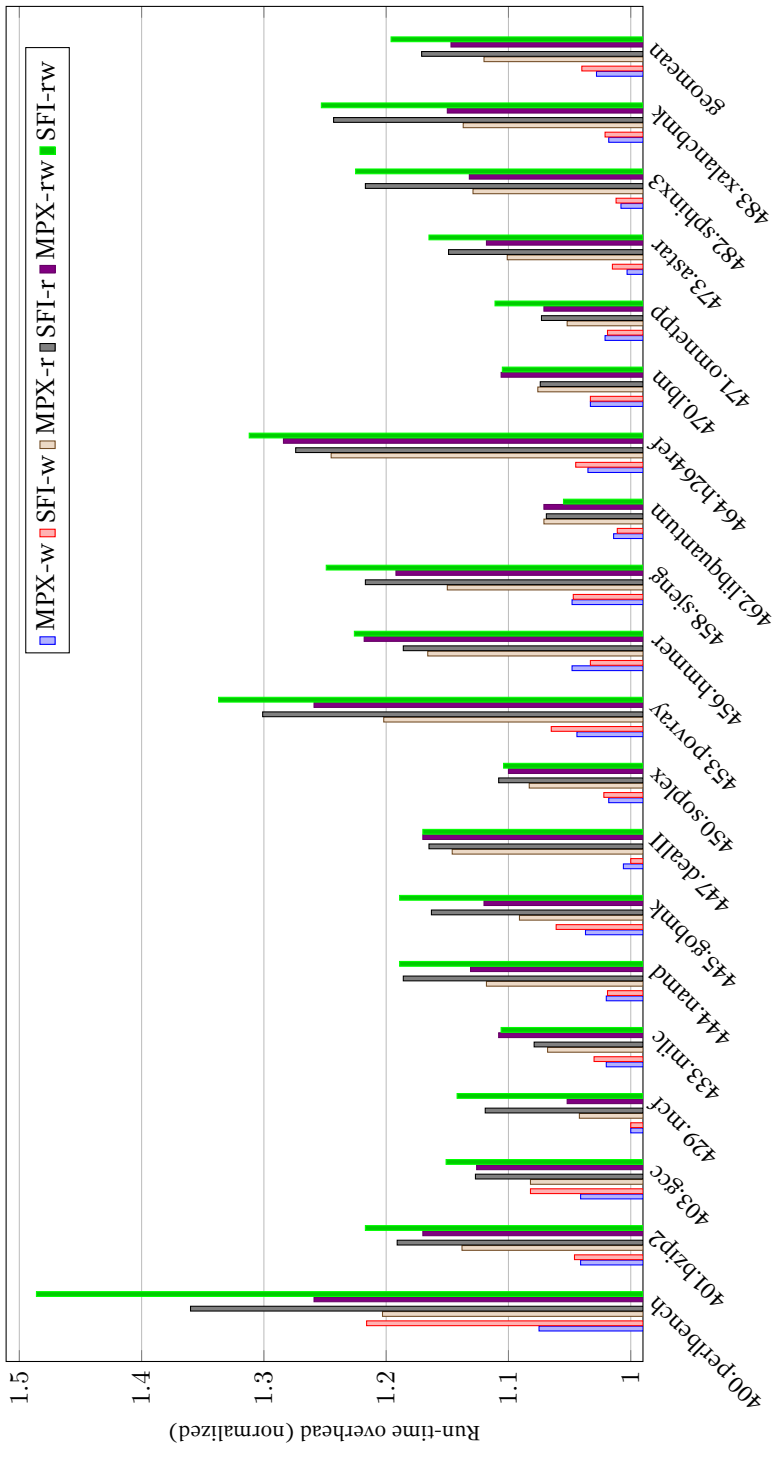


Figure 2.2: SPEC overhead for instrumentating all stores (-w), loads (-r) and both (-rw) for SFI and MPX, showing MPX introduced less overhead than SFI in most cases.

While applying memory isolation to SafeStack [114], a shadow stack implementation used in production and present in Clang [47], we opted for address-based isolation based on these results. SafeStack introduces no additional overhead on its own, as it simply replaces all stack loads and stores with accesses to the unsafe stack. We found that when applying MemSentry, SafeStack still introduced no additional overhead, and the results were identical to Figure 2.2.

All previous results assume a very small safe region in the case of crypt: a single native 128-bit (16 byte) value. Further experiments showed that in this case, most of its overhead is due to the initialization of round keys. Encryption of larger sizes increases linearly on top of this initial cost. However, we still observed an approximately 15x overhead when protecting a region of 1024 bytes.

2.6.3 Discussion

In this section, we look at further trade-offs for the different memory isolation techniques, evaluate the merits of each approach and, combined with the performance figures, give an overview of these techniques in practice. We hope this will help future researchers with deciding between such techniques, but we should note that all techniques are practical, depending on the situation (e.g., an older processor).

For address-based isolation techniques, MPX has many advantages over SFI. First of all, the performance of MPX is higher in almost all cases. It also provides a more flexible way of partitioning the address space, as bounds can be placed at any point in memory. Additionally, MPX deterministically informs the system of an invalid attempt, whereas with SFI the masked pointer may still be valid (in the nonsensitive partition). On the other hand, the biggest downside of MPX is that it requires a relatively new Intel CPU (Skylake architecture, released 2015). MPX also becomes much less favorable when many different domains are required, and because bounds must continuously be spilled to memory. Furthermore, in a situation with arbitrary bounds, where both upper and lower bounds must be checked, the overhead also becomes worse: our experiments showed it to be slightly worse than our SFI results. However, SFI cannot support such a situation at all without the need for far more expensive checks.

For domain-based solutions, we would expect MPK to have much higher performance than alternatives, based on our approximated benchmarks. However, the biggest issue with MPK is that it has not yet been released, nor have processors supporting it been announced. Until that time, the choice remains between VMFUNC and crypt. The cost of crypt increases linearly with the size of the isolated domain, whereas the cost remains constant for VMFUNC. This smaller granularity can also be an advantage: for smaller data (1–2 128-bit chunks) crypto is generally faster, and it does not require the data to be placed in separate pages. On the other hand, VMFUNC requires both fairly recent hardware (Intel Haswell or newer, available

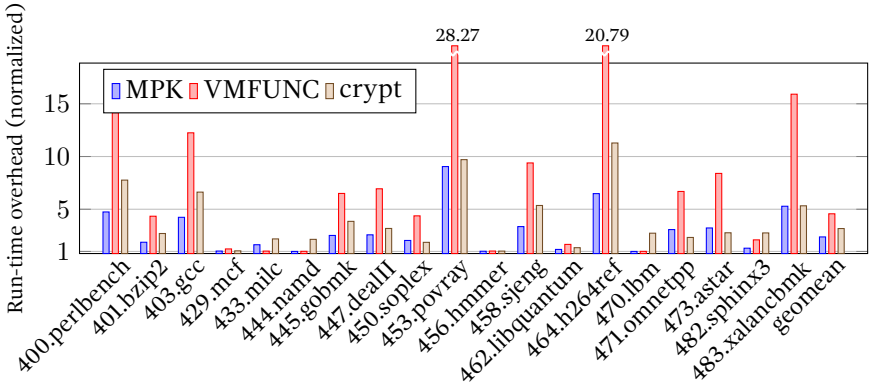


Figure 2.3: SPEC overhead for MPK, VMFUNC and crypt (on a single 128-bit chunk) when switching domains for every `call` and `ret` instruction, simulating a shadow stack.

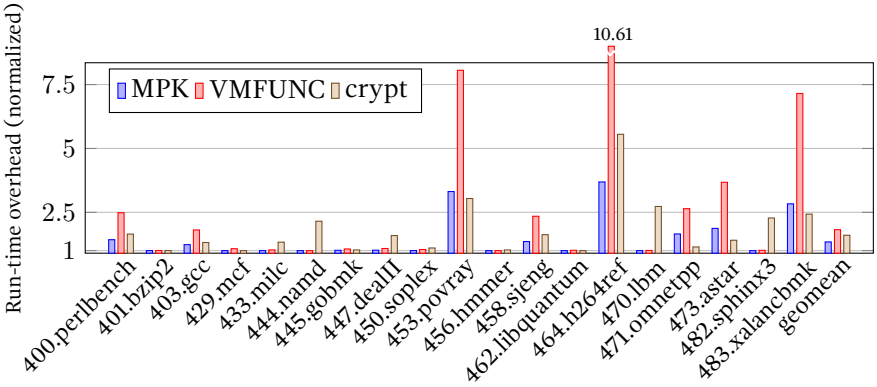


Figure 2.4: SPEC overhead for MPK, VMFUNC and crypt (on a single 128-bit chunk) when switching domains for every indirect branch.

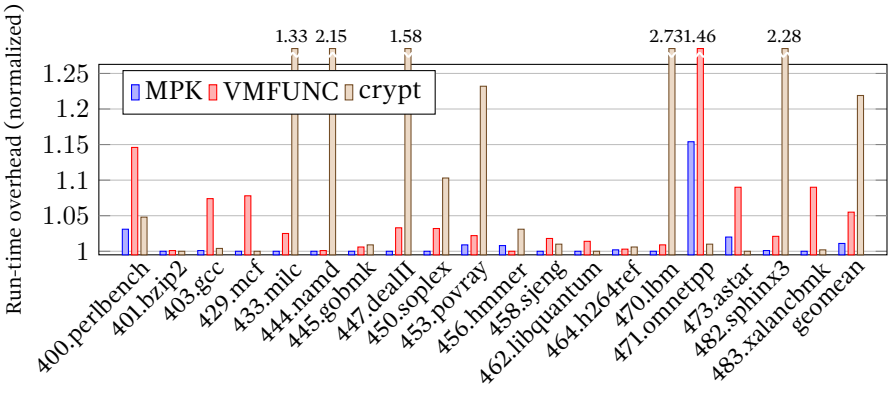


Figure 2.5: SPEC overhead for MPK, VMFUNC and crypt (on a single 128-bit chunk) when switching domains for every system call.

since 2013) and a (relatively small) piece of privileged code on the host: a modified hypervisor. AES-NI has been present in CPUs for far longer, beginning with the Intel Westmere architecture in 2010, and might thus be more widely and easier deployable. Finally, SGX, while an interesting technique useful for cloud environments, is not practical for the relatively lightweight isolation as discussed in this chapter.

When choosing between addressing-based and domain-based techniques, for MPX versus MPK, the optimal choice primarily depends on how often domain switches occur in practice (in other words, what portion of the instructions executed access isolated data). When this happens frequently, such as for every `call` and `ret` instruction, addressing-based approaches are more favorable.

2.7 Related work

Software-fault isolation has been actively researched over the past 20 years [70, 134, 211]. For instance, Native Client [184, 219] compiles programs written in C/C++ to verifiable sandboxed code, that run natively yet cannot execute arbitrary branches or memory accesses. LR^2 reduces SFI checks and achieves execute-only memory (XoM) by instrumenting instructions so that all pointers are masked [30]. Although LR^2 targets mobile devices and a particular application (resilient to leakage code randomization), the SFI optimization can be incorporated easily to MemSentry and implemented using hardware features of x86-64 such as MPX.

In terms of hardware-based isolation techniques, a lot of options exist across different architectures, many of which could be implemented as part of MemSentry. For instance, *segmentation* present in x86 (and dropped for x86-64) was used as an easy way of isolating memory [74, 211, 219]. ISBoxing [59] uses instruction prefixes to bound load/stores to a memory range, however, this significantly reduces the available address space. Similar features are present in other architectures, such as *memory domains* in ARM32 [226] (but removed from modern AArch64), which work almost identically to Intel's upcoming *MPK* feature for x86-64, except that the domain permissions can only be modified in supervisor mode. ARM also supports *TrustZone*, offering both a normal and secure world which are completely isolated, leveraged by TZ-RPK and Kenali [13, 192]. Usage of a *trusted platform module* allows pieces of application logic to securely run, and can leverage either hardware based solutions such as Intel TXT [136] or software-based solutions [135]. Sanctum [49] provides SGX-like features in hardware for memory isolation.

Several systems have been proposed to prevent stealing keys from memory [132, 166], including the use of *hardware-transactional memory* [83]. AMD's upcoming secure memory encryption (SME) encrypts all memory transparently, protecting it from outside analysis and cold-boot attacks. Secure encrypted virtualization (SEV) expands this to allow for per-VM memory encryption [102].

However, both SME and SEV do not help against local memory exploits. Isolating complete applications running in untrusted environments has been also studied. For example, Intel SGX can be used to protect applications running in untrusted clouds [17, 223], while other approaches are based on *virtualization* [38, 44, 126, 189, 218]. For instance, SeCage [126] offers domains that can be efficiently switched between using `vmfunc`, similarly to what was discussed in Section 2.3.1, and includes automatic application decomposition using points-to analysis. Both the decomposition of software and the protection of domain switches has been a major issue for such systems. Systems such as Readactor [53] and Heisenbyte [196] similarly use the EPT feature of VT-x to achieve XoM and destructive code-reads respectively. Finally, entirely new hardware features have been proposed to achieve memory isolation [206, 214].

We takes inspiration from a number of these approaches in how to leverage hardware-features to provide memory isolation. All of the aforementioned systems implement a custom protection scheme, supporting only a single hardware-feature, and often supporting only a single defense-system. In contrast, we provide a more general form of memory isolation on x86-64, and a general-purpose design supporting different isolation techniques for defense systems.

2.8 Conclusion

In this chapter we explored the importance of isolation in realizing software defenses. We reviewed a series of systems which rely on information hiding for securing some vital component for their operation. We argued that such probabilistic isolation is insufficient, and presented a design that can apply hardware features to provide *deterministic* isolation. We introduced MemSentry, a framework that can easily provide strong memory isolation to existing defense systems, and a practical evaluation of such hardware features.

We believe that MemSentry is useful for the community, since it provides a vital feature for many defense systems: the ability to properly isolate sensitive data from the rest of the process. Additionally, it helps further research towards this goal by offering a platform for testing hardware-supported isolation. MemSentry can be found at `http://github.com/vusec/memsentry`.

3 Delta Pointers: Buffer Overflow Checks Without the Checks

Despite decades of research, buffer overflows still rank among the most dangerous vulnerabilities in unsafe languages such as C and C++. Compared to other memory corruption vulnerabilities, buffer overflows are both common and typically easy to exploit. Yet, they have proven so challenging to detect in real-world programs that existing solutions either yield very poor performance, or introduce incompatibilities with the C/C++ language standard.

We present Delta Pointers, a new solution for buffer overflow detection based on efficient *pointer tagging*. By carefully altering the pointer representation, without violating language specifications, Delta Pointers use existing hardware features to detect both contiguous and non-contiguous overflows on dereferences, without a single check incurring extra branch or memory access operations. By focusing on buffer overflows rather than other vulnerabilities (e.g., underflows), Delta Pointers offer a unique checkless design to provide high performance while still maintaining compatibility. We show that Delta Pointers are effective in detecting arbitrary buffer overflows and, at 35% overhead on SPEC, offer much better performance than competing solutions.

3.1 Introduction

Almost 30 years after the Morris Worm famously used a buffer overflow bug in `fingerd`, such vulnerabilities are still rife in modern C/C++ binaries. Ever since, researchers have searched for ways to automatically detect and prevent the triggering of buffer overflows. Unfortunately, existing solutions suffer from either unacceptable overheads or poor compatibility. In this chapter, we add an interesting new point in the design space to detect both contiguous and non-contiguous buffer overflows, with less overhead than comparable solutions (35% performance overhead on SPEC CPU2006 and negligible memory overhead).

Aiming for a *practical* defense against the most prevalent attacks, we limit our focus to buffer *over*flow vulnerabilities whereby attackers can overwrite memory after the end of the buffer, rather than *under*flow vulnerabilities which are far less common [124][1]. On the other hand, since non-contiguous overflows are now more common than contiguous overflows in real-world exploits [140], simply surrounding all buffers with so-called red zones (as used by AddressSanitizer [186] among others) no longer suffices, since doing so prevents only contiguous ones. Instead, we strive for a solution that prevents all types of overflow and works "out of the box" on programs written in standard C code, simply by recompiling the code with a specific compiler flag.

Bounds checking is one of the oldest and most common defenses against buffer overflows. By recording the bounds of an object with each pointer, defenses can insert run-time checks to verify that the pointer still falls in the valid range upon dereference. Sadly, bounds checking is still costly due to metadata management, but especially because of the checks. Reading the bounds data from memory, verifying the validity of the pointer, and optionally branching if the pointer is temporarily out of bounds or missing metadata due to uninstrumented code all incur significant overhead. As a consequence, researchers during the past three decades have focused on improving both the performance of such systems and their compatibility with the standard. Although the solutions have significantly improved over time, state-of-the-art systems still suffer from compatibility issues and non-practical overheads. For example, the fastest contiguous and non-contiguous buffer overflow detector today, Low-Fat Pointers [66], still incurs over 50% performance overhead (SPEC CPU2006) and yet cannot automatically support arbitrary off-by-one pointers allowed by the C/C++ standard and required by many real-world programs [46].

To address these issues, we propose Delta Pointers, a new approach to detect buffer overflows. The Delta Pointers design is different from all existing approaches in that it implicitly invalidates pointers upon going out of bounds

[1] Anecdotally, although labels in the CVE database are not very precise, buffer overflows outnumber underflows by almost two orders of magnitude in such database: `https://cve.mitre.org/`

(providing good performance) and revalidates them when going back in-bounds (providing good compatibility). Specifically, we show that our solution is significantly faster than existing solutions while providing compatibility with the C standard (and we discuss remaining compatibility issues in detail). We ensure that any dereference of an invalid pointer leads to an automatic crash enforced by the hardware—similar to how a dereference of a kernel pointer in user space leads to a crash. As a result, Delta Pointers *eliminate any need for additional memory accesses or branches*, pushing the overflow check to the hardware and making pointer dereferences very efficient.

Intuitively, we utilize pointer tagging and ensure that any arithmetic on the pointer that makes it point beyond the buffer's upper bound, will result in setting the most significant bit in the 64 bit address, while any arithmetic that makes it point below the upper bound, will unset this bit. Prior to a dereference, we mask out all other bits of the tag. If the most significant bit is not set, the pointer is valid, but a dereference of a pointer where this bit is set will immediately crash the application because such addresses are non-canonical, triggering a fault in the processor's memory management unit.

Our approach makes minimal assumptions and is portable across 64-bit architectures. It works with existing hardware and most C/C++ programs out of the box (and we elaborate on possible issues in later sections). While pointer tagging itself provides good compatibility, we outline some challenges faced by practical implementations which have not been detailed by previous pointer tagging-based solutions [110, 113].

To summarize, our contributions are:

- A novel, fully automated, design to detect buffer overflows based on pointer tags that automatically invalidates out-of-bounds pointers.
- An analysis of the practicality of pointer tagging for arbitrary C/C++ applications.
- An LLVM-based prototype of our design, evaluated with respect to effectiveness and performance. Our results show that Delta Pointers can detect the dominant class of spatial memory error vulnerabilities with competitive compatibility and overheads (35% slowdown on SPEC and negligible memory overhead).

3.2 Background

In C and C++, a programmer can allocate a *buffer object*, which is a sequence of bytes somewhere in the address space. For instance, variables on the stack are implicitly allocated on function entry and a programmer can use `malloc` or `new` to explicitly allocate a buffer object on the heap. These objects are referenced through *pointers*,

which point to the memory location of an object, or a location therein. The C standard [97] specifies that pointer arithmetic only produces defined behavior if the resulting pointer points inside the same object or at the element directly past its end. In practice, compilers produce invalid *out-of-bounds pointers* without warning. Without any security measures, such pointers can be used to access any object in the address space, regardless of which buffer the original pointer pointed to.

For example, a call such as `ptr = malloc(16)` allocates a new object on the heap and returns a pointer to its beginning. An offset of exactly 16 would yield a pointer to the end of the object, which is valid but results in undefined behavior upon dereference. Any dereference of an index smaller than 0 (e.g., `ptr[-10]`) is an underflow, and any dereference of index 16 or larger is an overflow. It is common for programs to (benignly) create pointers pointing outside the referent object [46]. However, if malicious users can lure the program into dereferencing such pointers as a result of a buffer overflow vulnerability, they may be able to leak or corrupt sensitive information, for instance to divert control flow.

Overflow detection To detect buffer overflows during the execution, existing spatial memory safety defenses insert run-time checks in programs. Such *instrumentation* consists of one or more checks, in the form of branches, on either pointer dereferences [65, 66, 87, 113, 150, 158, 186] or pointer arithmetic (e.g., `ptr2 = &ptr[offset]`) [6, 100, 221]. Moreover, the metadata describing object bounds must be recorded, propagated and retrieved numerous times, often from memory [87, 100, 113, 150, 158, 186, 221]. Given the many extra branches and memory accesses required, the run-time checks can introduce significant runtime overhead.

As a result, the reduction of the number of branches and (especially) memory accesses has been the motivation for most prior research in spatial memory safety. Early defenses stored all their metadata in memory, for instance recording the base and size for every object in the program [100]. More recent defenses limit such in-memory metadata to only pointers that reside in memory [150] or to only the lower bounds of objects [113]. Alternative designs avoid storing metadata in memory altogether by manipulating the memory layout to implicitly encode the size and base of the object in the address of its memory location [6, 65, 66].

The need for (expensive) branching has also been a target of optimizations. Traditional spatial safety defenses require two branches: one for the lower bound and one for the upper bound of an object [150]. More recent defenses have suggested reducing pointer validation to a single branch by enforcing predictable (but wasteful) power-of-two alignment on allocated objects [6]. Besides pointer validation, existing defenses also require additional branches for compatibility with common real-world scenarios, such as branching on temporarily out-of-bounds pointers [6, 32, 221] and branching on uninstrumented pointers with NULL metadata (e.g., in the case of uninstrumented shared libraries) [113, 150].

In this chapter, we focus on the most common class of spatial safety errors (*buffer overflows*) and investigate whether minimizing the number of extra branches and memory accesses (and the overhead) to detect overflows is possible. We start our analysis with a simple adaptation of state-of-the-art solutions. Specifically, we developed an adaptation of SGXBounds [113] that only stores the upper bound of an object in the high bits of its base pointer and uses a (branching) check on each memory access to compare the two parts of the pointer and detect (only) overflows. This design eliminates extra memory accesses, but still requires branching (for both pointer validation and support of uninstrumented modules) and incurs 48% (SPEC) overhead in our experiments. With Delta Pointers, we present a new design that can eliminate extra branches and memory accesses altogether, while retaining the same security guarantees and incurring only 35% (SPEC) overhead. In other words, removing the branches reduces the overhead by roughly 25%.

3.3 Threat model

We consider an attacker able to exploit a buffer overflow by feeding malicious inputs to a given vulnerable user program. We assume the attacker can repeatedly interact with the program, and the program is automatically restarted in case of crashes caused by failed exploitation attempts. Our goal is to detect user-space exploitation of arbitrary buffer overflows when memory is either written to of read from, protecting both integrity and confidentiality.

3.4 Delta Pointers

Delta Pointers eliminate the need for branching checks by encoding the out-of-bounds state of a pointer in the pointer itself. Rather than relying on expensive instrumentation, we use hardware memory protection mechanisms to *implicitly* invalidate pointers that go out-of-bounds, and revalidate them when they go back in-bounds. Upon dereference, out-of-bounds pointers automatically trigger a fault. This is possible by encoding, in every pointer, a *tag* that describes the distance to the end of the memory object. This scheme aims to translate the buffer overflow detection problem into an arithmetic overflow detection problem, which can be dealt with more efficiently thanks to (i) detection offloaded to the hardware, (ii) efficient load/store and pointer arithmetic instrumentation with no branching or memory accessing instructions, (iii) no extra branches required to support common compatibility features, such as temporarily out-of-bounds pointers and uninstrumented pointers.

The absence of memory accessing instructions and out-of-band in-memory metadata provides two additional benefits other than good performance. First, the

lack of in-memory metadata ensures a compact memory footprint (Delta Pointers introduce negligible memory overhead). Second, using only in-pointer metadata eliminates any need for synchronization across threads for metadata management, ensuring scalability and thread-safety in multithreaded applications.

Pointer encoding To enable our design, we make use of pointer tagging both to implement implicit out-of-bounds pointer invalidation and to ease propagation of metadata inside and across contexts. Specifically, each pointer is tagged with the current distance from the pointer to the end of the object, called the *delta tag*, and an *overflow bit*. As the distance from the current pointer to the end of the object changes during pointer arithmetic, the delta tag is kept consistently up to date. The overflow bit indicates whether the pointer is out-of-bounds, and is implicitly set during delta tag updates. To facilitate the implicit management of the overflow bit, the delta tag is encoded as the negated remaining distance, as we will explain later. Figure 3.1 illustrates our encoding scheme: the uppermost bit is the overflow bit, followed by the delta tag. This encoding allows for 32-bit tags, limiting addresses to 32 bits similar to prior tag-based schemes [113]. However, in contrast to prior schemes, in our design this division of bits can be changed arbitrarily depending on the application; the address space need not be limited to 32 bits. Instead, it allows a trade-off between the maximum object size and the address space size: as one increases, the other decreases, both by a factor of two.

The key insight exploited by our encoding scheme is that, by updating the delta tag alongside the pointer itself (which can be done efficiently), the state of the overflow bit is managed implicitly. Figure 3.2 shows the state of the bits during several operations on the pointer. The allocation on the first line creates a new pointer and initializes the delta tag to $-object\_size$. When adding an offset to the pointer, the addition is replicated on the delta tag, as shown on the second line. On the third line we do the same, this time causing the pointer to go out-of-bounds: the accumulated offset added to the delta tag is equal to the encoded object size. The pointer now points to the end of the object and the distance towards the end is 0. The carry bit of the addition on the delta tag sets the overflow bit implicitly. The fourth line shows that the same mechanism can bring a pointer back in bounds. When we subtract 1 from the entire upper part of the pointer (the delta tag and the overflow bit), it returns to the same state as in the second line.

Whenever the program dereferences a pointer, we first apply a bitwise AND operation to mask out the delta tag and create the regular untagged pointer the CPU expects. By maintaining the overflown state of a pointer in its upper bit we eliminate the need for an explicit (branching) check when dereferencing the pointer. Instead, we leave this bit intact by not masking it out with the delta tag as illustrated in Figure 3.3. Through this approach, we delegate the check to the memory management unit (MMU) of the processor: a pointer can only be used if it is in a canon-

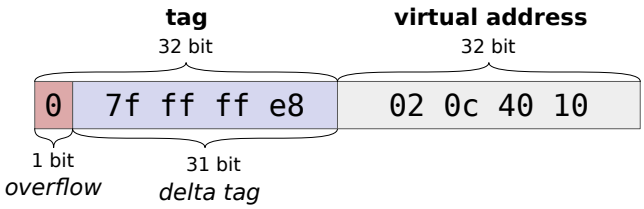


Figure 3.1: The encoding of Delta Pointers (tagged pointers). It points to an object on the heap of 24 bytes. The delta tag encodes the distance from the current pointer to the end of the object. The value of the delta tag is calculated as $-distance$ (here -24, in two's complement). The most significant bit is only set if the pointer is out-of-bounds.

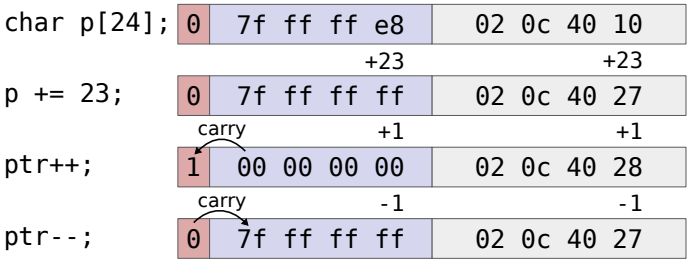


Figure 3.2: Examples of how the pointer tag is updated during pointer arithmetic: every operation on the address (lower bits) is also performed on the tag (higher bits). The most significant bit, indicating whether the pointer is out-of-bounds, is implicitly set and unset during these operations.

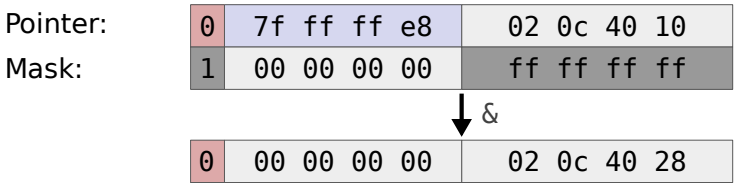


Figure 3.3: Before using the pointer to access memory, we have to mask out the metadata to create a valid pointer. By using a bitmask with the most significant bit, we keep the overflow bit of the pointer intact. If the pointer was overflown, and thus had its overflow bit set, this bit will still be set in the resulting pointer. This is an invalid (non-canonical) pointer, causing a run-time error.

ical form, where the 16 most significant bits are sign-extended [96]. For pointers in user space, this equates to 16 zero bits, since the upper bit of an 48-bit pointer is reserved for kernel space in most operating systems. A set overflow bit therefore makes the pointer uncanonical, causing the MMU to generate a fault. The underly-

ing assumption is that bitwise masking operations, combined with arithmetic operations on pointer additions, are highly optimized on a microarchitectural level, and are therefore faster than explicit checks with branches. Even though on modern architectures the branch predictor is heavily optimized, branches still incur overhead.

Listing 3.1 shows the translation of these two concepts into code instrumentation. ① creates the delta tag in the correct format and places it in the upper bits of the pointer. At ② we replicate the addition that happened in the lower bits to the upper bits. Finally, ③ performs masking, leaving the pointer and the overflow bit intact. Because the delta tag is part of the pointer itself, it is automatically propagated across contexts, and will not result in any memory accesses. Moreover, ② and the pointer arithmetic can be combined to a single operation: `nptr = ptr + (user_input + (user_input << 32))`, even down in the output assembly.

NULL pointer protection The NULL pointer constitutes a special case of a pointer that is not derived from any object, but still provides an (often neglected) attack surface for spatial errors. Exploits for such bugs typically add an attacker-controlled offset with the value of the target location to grant the attacker arbitrary memory read or write capabilities [106]. For Delta Pointers, we set a delta of 1 on all NULL pointers, thus replacing them with a value of `0x7fffffff00000000`. This will cause any dereference of a pointer derived from NULL to trigger a fault and hence detection.

Thread safety Delta Pointers are inherently safe with respect to racy pointers. Added instrumentation on memory operations and pointer arithmetic consists of arithmetic operations that operate on registers, and no additional memory accesses are introduced. Modified pointers are written to memory in a single atomic operation (on most architectures), causing the pointer tag to always be consistent with its corresponding address. An existing race condition may be influenced by the introduction of additional operations on either end, but this is an inherent problem of racy pointers, not a race condition introduced by Delta Pointers.

Address space reduction Since pointer tagging-based defenses use part of a pointer to encode metadata, they reduce the amount of bits left for addresses. Because this limits the addressable virtual address space, ASLR entropy is reduced as well. For instance, Low-Fat pointers [65] reserves a large virtual memory region for each object size class, and SGXBounds [113] encodes two 32-bit addresses side by side in each pointer, thus not fully utilizing the 36 bits of entropy offered by SGX. Delta Pointers similarly limit addresses to 32 bits by default (but configurable on a per-application basis). Although ASLR has a wider scope than these schemes (e.g., use-after-free bugs), it can only provide probabilistic

```
void *foo(int user_input) {
  char *ptr = malloc(16);
  delta_tag = (1 << 31) - 16;                                    ①
  ptr |= delta_tag << 32;

  char *nptr = &ptr[user_input];
  nptr += (uintptr_t)user_input << 32;                           ②

  tmp = nptr & 0x80000000ffffffff;                               ③
  *tmp = 'a';
}
```

Listing 3.1: C program instrumented with Delta Pointers. Instrumentation is shown as pseudocode on the highlighted lines. ① sets the delta tag, ② updates the delta tag alongside the pointer, and ③ strips out the metadata (but not the overflow bit) before a memory access.

safety at best, whereas Delta Pointers provide deterministic (spatial) memory safety guarantees on the upper bound. Because of its probabilistic nature, ASLR has proven to be easily circumventable by memory massaging [81, 156] or side channels [29, 76, 82] whereas this is not possible for deterministic defenses such as Delta Pointers. Moreover, the impact of address space reduction is limited in certain application domains. As an example, similarly to SGXBounds, Delta Pointers are well suited to run arbitrary programs in SGX enclaves, which only support 36-bit addresses currently. By encoding metadata in pointers rather than in memory, the only memory overhead of Delta Pointers is that of added code (which is negligible), even amounting to an advantage over, for example, shadow memory-based schemes in SGX where virtual memory is limited.

Our Delta Pointers prototype uses 32 bits to address a 4 GB address space. The remaining 31-bit delta tag allows for a maximum allocation size of 2 GB. This split is configurable and can be tweaked depending on the application: a program that allocates a lot of smaller objects can have a bigger address space. For instance, without any address space reduction, an application would have 47 bits of address space and 16-bit delta tags, allowing for only 64 KB objects. For Delta Pointers we did all evaluation with 32-bit addresses and 31-bit delta tags because our experimentation shows this achieves a good compatibility with a large set of complex real-world programs under realistic workloads.

3.5 Pointer tagging

Delta Pointers are an instance of pointer tagging to efficiently store metadata per pointer, yielding a design that has low overhead for lookups, negligible memory overhead and automatically works with concurrent programs. These benefits do not come for free: modifying the representation of pointer introduces various chal-

lenges regarding compatibility and performance. These challenges are not limited to Delta Pointers, but are generic in nature and must be dealt with by any defense that encodes metadata in pointers.

In this section, we discuss the challenges that need to be addressed by pointer tagging based approaches, with the aim of inspiring and assisting future research based on this increasingly popular technique. We also present the solutions to the listed problems which are implemented in Delta Pointers, showing that high compatibility with complex real-world applications is possible in the presence of tagged pointers. We discuss four challenges with pointer tagging. First, adding arbitrary metadata in pointers requires careful implementation of pointer operations to conform to the C standard and implicit assumptions of C programs on its implementation. Second, compilers make certain transformations that make pointer identification harder, in particular during optimizations. Third, defenses must deal with pointers that do not have metadata due to optimizations or uninstrumented libraries. Finally, the speed of decoding tagged pointers is influenced by micro-architectural properties.

3.5.1 C pointer operations

The presence of metadata tags in pointers means that, without additional measures, the integer comparisons and arithmetic operations may no longer accurately implement the semantics defined by the C standard [97]. Moreover, while the C standard generally attempts to leave data representation to be defined by the implementation, in practice many C programs make assumptions that go much further than the guarantees provided by the standard [46].

Additional instrumentation, in particular targeted removal of pointer tags, is required to achieve compatibility of arbitrary pointer tags with existing programs. Without this instrumentation, a program would make different computations or control flow decisions based on tagged pointers rather than regular ones. This can cause the program to crash or produce incorrect results. The problem only arises, however, when tags encode *per-pointer* metadata. This means that two pointers pointing to different locations in the same memory object may have different tags, causing operation semantics to change. Solutions using *per-object* metadata, such as SGXBounds [113], do not experience this problem, since they do not modify pointer tags after object allocation.

Pointer comparison In the C standard, the outcome of comparison operators on pointers is only defined if the pointers either point to the same object or are part of the same aggregate object. Although other cases are explicitly left undefined, the standard does state in general that “the result depends on the relative locations in the address space of the objects pointed to”. In practice, however, programs often

assume that pointers are implemented as simple integers and expect a total order on them, for instance for sorting.

When tagging pointers, this assumption no longer typically holds, especially for dynamic tags. Tags should thus be masked away before comparing pointers to avoid an incorrect result. For Delta Pointers we mask away the tag including the overflow bit. This is safe, since the masked pointers are only used for comparison and never dereferenced, so no buffer overflow checks are needed.

Pointer subtraction The C standard allows subtraction of pointers to the same array object. These are used to compute distances between objects in the address space. While normally implemented as an integer subtraction instruction, this may yield incorrect results in the presence of different pointer tags. The tags of subtracted pointers must therefore be masked away. Some programs even subtract pointers to different objects, thus violating the standard and making assumptions on the memory layout. However, these applications are still supported when tags are removed. Note that arithmetic optimizations by compilers may rewrite complex expressions involving pointers in such a way that these cases are introduced as well, as also discussed below. All these scenarios are supported by our Delta Pointers implementation.

Pointer alignment Non-conforming programs often use bitwise operations on pointers to detect or enforce alignment properties. Such alignment works correctly for Delta Pointers without any additional instrumentation, since it can only round pointers down, increasing the distance to the end of the object. This will not lead to false positives since applications themselves must assume that the pointer remained unchanged and thus did not increase the distance to the end of the object (which is what Delta Pointers enforce).

3.5.2 Compiler support

Pointer tagging requires type information to be able to distinguish pointers, which, in turn, requires the defense to be implemented at the compiler level. Code instrumentation can be done at different compilation stages, in particular before and after optimizations. Instrumenting before optimizations is easier because then the code has not been transformed to patterns that hinder static analysis. Instrumenting all pointer values, however, severely handicaps optimizations: the instrumentation casts pointers to integers in order to add, modify, or remove the tag, breaking alias analysis. Hence, to attain high performance, tagging must be applied after optimizations. This raises several issues that need to be addressed by pointer tagging implementations. When left unaddressed, these issues can either cause the instrumentation passes to fail due to unexpected inputs, or have them produce in-

accurate instrumentation code that corrupts the program state at runtime. To the best of our knowledge, our Delta Pointers implementation is the first to provide wide compatibility with complex source code (e.g., SPEC CPU2006) due to the solutions described below.

Pointers as integers Modern compilers such as LLVM implement optimizations that produce arbitrary typecasts from pointers to integer types. Hence, static analysis is needed to determine which values are pointers in optimized IR. For dereferencing operations, the problem does not occur because the pointer value must always be cast to a pointer type prior to dereference. Comparison, however, is defined on both integers and pointers, and subtraction only on integers. Therefore, a pointer value that has an integer type need not be typecast prior to these operations, making it unclear whether masking is needed. This problem can be solved using use-def chains [146] to trace back the origin of the pointer to its definition (an allocation or function parameter) or a load from memory. Definitions contain the original variable type, which can be used directly to determine if the value is a pointer. Pointer loads from memory, however, can be transformed arbitrarily to integer loads by optimizations. If the loaded pointer is never dereferenced, the use-def chain does not provide information on its actual type, and the value is not masked, producing incorrect behaviour. The following example illustrates such a case:

```
int **a = ...;              // uint64_t *a;
int **b = ...;              // uint64_t *b;
diff = *b - *a;             // diff = *b - *a;
```

The pointers pointed to by `a` and `b` are never dereferenced; their values are only subtracted and the result is stored in memory. This allows the compiler to remove the (implicit) type casts of `*a` and `*b` to integers, effectively producing the code on the right, which alters the type information of the values in memory.

This problem can be partially solved in two ways. First, metadata describing the types can be added to values being loaded/stored before running optimizations. Certain optimizations, such as type-based alias analysis (TBAA) [63] already add such information, which can be reused. However, such metadata can potentially be removed by subsequent optimizations, and might thus not be complete. The second solution is to trace back the pointee type through the use-def chain of the address that is loaded, using nested type inspection to find a pointer type. This on itself works reasonably well since the folding of memory loads and type casts is often very localized, requiring only limited backtracing of the use-def chain to find the original pointer type. Our Delta Pointers implementation uses a combination of these solutions, which our experiments show is complete for all the tested programs.

Unions and pointer expressions A union value of a pointer and an integer may produce a typecast from a pointer to integer. The operation can either be operating on the integer value or it can be operating on the pointer value but preceded by a typecast because the operation happens to only be defined on integers. For example, bitwise operations are only defined on integers but are also used for pointer alignment. Furthermore, compilers transform expressions involving pointers in ways that sometimes produce unexpected pointer operations. For example, consider $(b - a) \times 4$ where a and b are pointers. This should be caught as a regular pointer subtraction as per Section 3.5.1, but the compiler may transform this to $b \times 4 + a \times -4$, thus introducing pointer multiplication and then addition. We observed similar cases in SPEC CPU2006 involving division, remainder, bit shifts, bitwise OR, and bitwise XOR, all on pointer operands. The case above is easily solved by masking the pointer operands of multiplications, but the tag must be preserved when the result is a pointer that may later be dereferenced in order to do a bound check. In other words, the decision whether to mask the pointer operand of an arithmetic expression thus depends on whether the resulting expression is indeed used as a pointer (i.e., it is dereferenced). This is determined by traversing its def-use chain until such a use is encountered.

Although the combination of the above solutions theoretically does not cover all code patterns expressible in compiler IR, it works very well in practice. For instance, our Delta Pointers implementation correctly identifies all pointers in the C/C++ SPEC CPU2006 benchmark suite after full-fledged optimizations. Our design significantly improves compatibility with respect to state-of-the-art pointer tagging implementations such as SGXBounds [113], which forcibly omits 6 out of 19 SPEC benchmarks due to incorrect pointer identification.

3.5.3 Coverage considerations

Many pointer tagging applications assume that each dereferenced pointer is tagged with metadata. For example, SGXBounds stores a pointer in the tag that is dereferenced to retrieve further metadata. When a pointer misses metadata, a `NULL` pointer is dereferenced and the program crashes. However, reaching complete coverage is not always possible or even desired. For instance, optimizations by the defense may omit instrumentation on pointers that only have safe uses. When these pointers are used at the same site as unsafe pointers, the instrumentation at that site cannot assume the presence of metadata in the pointer. Furthermore, uninstrumented libraries may generate untagged pointers, and cannot handle tagged pointers passed in library call parameters.

A robust pointer tagging-based defense should therefore be designed to deal with missing metadata. Unfortunately, most existing defenses require extra branches to deal with missing metadata. Alternatively, and even less desirably,

such defenses must instrument all libraries and disable aggressive optimizations. In contrast, Delta Pointers are robust by design against missing metadata because the zeroed metadata is not dereferenced as a pointer. It effectively treats missing metadata as a very large distance to the end of the object, larger than the object can possibly be, therefore simply disabling the bound check in a situation where metadata is missing (which is the expected behavior).

Uninstrumented libraries are, in fact, part of a general problem with pointer tagging: there is always a *protection boundary* at which marshalling must occur between tagged and untagged pointers. Even if all libraries are statically linked and instrumented, the boundary is only moved to system calls that cannot operate on tagged pointers (unless even the OS kernel is rewritten). Ideally, pointers handled by the protected application are always tagged and pointers outside the boundary are not. In other words, the storage of pointers can not be *shared* between protected and unprotected code. A complex example is `std::list::push_back` in an uninstrumented `libstdc++`. The implementation of this function is defined in a header file and therefore resides in the protected executable. It calls a library function that takes a generic node type for doubly linked lists as a parameter. This data structure is created in `push_back`, which is instrumented because it resides in the protected executable, and thus tags the node's `next` and `prev` pointers with metadata. Code inside `libstdc++` does not mask the pointers before dereferencing and crashes if these pointers are tagged.

Defining a protection boundary requires a set of rules that specify when to add or remove tags to pointers. Rules for removing tags are mostly universal, but rules for adding tags depend on the specific defense being implemented.

3.5.4 Performance considerations

The minimum instrumentation required to implement pointer tagging consists of tagging and masking. These are bitwise operations that are fast in modern, pipelined processors. However, they may still have an effect on the pipeline and increase register pressure when the masking constant does not fit in an immediate operand (as is the case for 64-bit masks on x86-64). For this reason, recent hardware designs are shipping with built-in support for *MMU-based masking*: AArch64 supports virtual address tagging [7] which introduces a top byte ignore option for the hardware to ignore the upper 16 pointer bits during dereference, specifically for the purpose of encoding metadata. Oracle's SPARC-M7 architecture [160] can even ignore up to 32 bits of metadata. This completely eliminates the need for software-based masking and paves the way for highly efficient encoding of per-pointer metadata. In other cases, efficient masking can be done with fewer restrictions on the bitmask width than on x86-64. For instance, ARM64's AND instructions support efficient variable-length encoding of certain 64-bit

immediates, including the bitmask required by Delta Pointers.

Another performance consideration of pointer tagging solutions is protection against metadata corruption. Arithmetic on a pointer may overflow into the metadata bits, allowing an attacker to bypass the implemented defense. SGXBounds experiences this problem, and thus needs to move the metadata out of a pointer before every pointer arithmetic instruction and move it back in afterwards, incurring significant additional overhead. Delta Pointers do not need special treatment here: a pointer overflow will also overflow the delta tag into the overflow bit, correctly invalidating the pointer.

3.6 Implementation

We have implemented a prototype of Delta Pointers for Linux on the x86-64 architecture on top of the LLVM compiler infrastructure [118] (version 3.8). The code consists of 3,749 SLOC of LLVM C++ passes, which add the instrumentation described in Sections 3.4 and 3.5. An additional 846 SLOC make up runtime and helper libraries, including a static library that shrinks the address space of the process to make room for tags in pointers. The code is open source.[2]

In order to harden an existing program with Delta Pointers, the programmer adds compiler flags that invoke our passes during the compilation of source into the binary, and during linking to link in our static library. The resulting binary is then run through a post-processing script to shrink its address space. The resulting binary can then run as-is, raising an error upon detection of an out-of-bounds memory access. Dereferences of out-of-bounds pointers will cause the MMU to trigger a general protection or stack segment fault, which Linux will deliver to our process as a segmentation fault or bus error. We install a signal handler to distinguish such cases and report an appropriate error.

3.6.1 Address space reduction

User-space pointers in Linux are 47 bits. We limit this to 32 bits to support 32-bit tags. Only changing the allocator is not sufficient for this purpose, as the kernel maps the stack and loader at high memory addresses. The loader itself will also run code performing allocations before we get a chance to insert code. A kernel patch is the most straightforward way to limit the address space for mappings, but provides poor portability. Instead, we use the approach of Mid-Fat Pointers [110] to limit the address space in user mode: First, we prelink the binary, loader, and any shared libraries used by the program at locations that fit 32 bits. Then, during program startup, we move the stack and thread-local storage down in the address

[2] `https://github.com/vusec/deltapointers`

space. Finally, we reserve the memory area above 32-bit with an anonymous non-reserved mapping to avoid subsequent allocations in this address range.

3.6.2 Instrumentation

Listing 3.1 in Section 3.4 describes the instrumentation added by our LLVM passes. We apply these changes after optimizations, including link-time optimization (LTO), so that these optimizations are not hindered by our inserted pointer-to-integer casts. A final optimization pass performs optimizations such as constant folding on the added instrumentation.

For dereferencing instructions, we consider LLVM's `load` and `store` instructions and memory intrinsics (which cover calls to the `memcpy` family). For memory intrinsics, we update the pointer tag with the size of the dereferenced memory range minus one so that the highest dereferenced pointer is checked. The size is truncated to the maximum object size of 31 bits to also check very large sizes caused by implicit casts from signed to unsigned integers.

3.6.3 Coverage

Finding all heap allocations Detecting overflows for all buffers requires identification of all memory allocations so that the resulting pointers can be tagged. Stack allocations and globals are trivially identified by their unique representation in LLVM, but heap allocations are performed by calls to memory allocator functions. We currently support all allocation functions in the C and C++ standard libraries. Custom allocators that preallocate a pool of memory using `malloc` or `mmap`, however, must expose allocation function names and the position of their size arguments to the Delta Pointers implementation in order to support per-object buffer overflow detection (otherwise only per-pool overflows can be detected). This is the case for *nginx* which we support by adding its custom pool allocation functions to our predefined list.

Pointer marshalling at library calls As explained in Section 3.5.3, making sure that all pointers in the protected executable have a tag requires a set of rules that define how pointers tags are added and removed at the protection boundary. For Delta Pointers, we have opted to place this boundary at the level of dynamic library calls in order to provide portability and maintainability, preserving the ability to update libraries without having to recompile all protected programs that use these libraries.

Pointers passed as parameters to library calls are masked in the same way as dereferencing operations, so the overflow bit is left intact. This design even prevents an attacker from using a vulnerable library function to dereference an already

out-of-bounds pointer (however does not protect against out-of-bounds dereferences if the library code adds an offset to the pointer). Nested pointers in data structures, such as those used in `std::list::push_back`, are stripped of metadata when a pointer to the data structure is passed to a library function. The tags are not restored after the function call, so these pointers remain unprotected inside the protected executable as well, thus assuming that the library implementations handling the data structures are safe. No tag is added back to the pointer, since it is not known how the function alters pointers in the data structure. Although this introduces untagged pointers in the protected executable, all functions that operate on the data structure are in fact implementations of standard library functions in header files. These functions are in fact outside the protection boundary, only included in the executable for optimization reasons (inlining). The only other case requiring similar instrumentation is `std::string::operator+=`.

Finally, data pointers returned by library functions are tagged to offer protection inside the executable. In particular, the rules specify how the distance from the returned pointer to the end of the referent object can be inferred from the call parameters. We have analyzed all functions in the C and C++ standard libraries and identified six categories:

- **copy**: Copy the tag of an argument. E.g., `strdup`.
- **diff**: $ret_{tag} = param_{tag} + (ret_{address} - param_{address})$. E.g., `strchr`.
- **static**: Tag is constant, size inferred from return type. E.g., `fopen`.
- **strlen**: $ret_{tag} = strlen(ret_{address})$. E.g., `getenv`.
- **strtok**: Special case for `strtok`: replace its implementation with a version that maintains the current end-of-object distance in a global variable.
- **noarith**: Disallow pointer arithmetic by assuming an object size of 1 byte. This is used for opaque return types such as that of `opendir`, whose returned pointer is not dereferenced inside the executable itself but only passed as a parameter to library calls.

Using these categories, we are able to instrument 99.7% of all dereferenced pointers in SPEC CPU2006. The remaining 0.3% are all related to shared state between protected code and unprotected library code. We verified through manual inspection that these cases can either be fixed with static analysis (e.g., `argv`) or can safely be ignored. An example of the latter case are C++ VTables which contain code pointers to virtual methods of objects. Each object stores an object to its VTable, which if instrumented cannot be dereferenced by libraries. The table structure is, however, an implementation detail of the compiler and not transparent to the programmer. All accesses to VTables are compiler-generated and can therefore be assumed to be in bounds (not accounting for compiler or type-confusion bugs).

Our Delta Pointers implementation therefore omits tags on VTable pointers in favor of compatibility. Note that no checks need to be inserted on pointers without metadata, which Delta Pointers support by design.

3.6.4 Optimization

Some existing bounds checkers implement analyses that identify *safe* memory accesses, which are statically known to be in-bounds. Unfortunately, these optimizations are not directly applicable to Delta Pointers, because the bounds checks are implicitly performed at pointer arithmetic sites. Masking instrumentation can only be removed from a memory access if none of the possible pointer values can have a tag, meaning that any instrumentation that adds or modifies the tag must first be removed. This is not always safe to do, for example when the same allocation is also passed to a function that does a possibly out-of-bounds memory access.

We use static analysis to find allocations, propagation sites (pointer arithmetic) and masking sites (loads/stores) that do not need instrumentation. In particular, we use LLVM's scalar evolution (SCEV) analysis [69] to trace back pointer bounds from loads and stores and omit instrumentation where the pointer can be proven to be in bounds. The analysis records the distances to the object start and end $(0, size)$ at each allocation site. These bounds are then propagated over the def-use chain of the allocation. At a propagation site, the offset is added to the recorded start distance and subtracted from the end distance. At a load or store, the recorded distances are used in combination with the dereferenced number of bytes to check if the dereferenced pointer is in bounds. Instrumentation is omitted on pointers produced by allocations or propagation sites that are only dereferenced in bounds. Masks are also omitted from dereferenced pointers that do not contain a tag because of optimizations. Note that since we use SCEV analysis, which represents IR operations as expressions, both $size$ and pointer offsets need not be constants, thus allowing for aggressive optimizations.

Another optimization we perform is the *hoisting* of bounds checks out of loops. Consider a simple loop that requires a bounds check in each iteration:

```
for (i = 0; i < 10; i++)
  buf[i] = 'x';
```

It is easy to see that this check only needs to be performed once on `buf[9]`. We use the same SCEV analysis as described above to infer the maximum value at compile time and insert a dummy load before the loop to trigger a fault if the computed offset is out-of-bounds. Instrumentation on the operations inside the loop is removed. This optimization can only be performed if the pointer `&buf[i]` does not escape the loop body. Unfortunately, LLVM rewrites the exit condition in the above loop to `i == 10`, thus making `&buf[10]` the maximum value of the pointer inferred by SCEV. Although the pointer has this value after the loop, it is never ac-

tually dereferenced. We have implemented a simple pattern detection to support this case, but the problem still exists for complex loops found in real-world programs such as in the SPEC benchmarks. We consider this a limitation of the SCEV implementation and therefore out of scope for this work.

3.7 Evaluation

To evaluate Delta Pointers we look at both their performance and security. To study the performance we benchmark SPEC CPU2006 and other real-world applications. We then evaluate the effectiveness of Delta Pointers by determining if they mitigate a number of recent CVEs reported by related work.

3.7.1 Runtime performance

We have evaluated Delta Pointers on the C and C++ programs of the SPEC CPU2006 benchmarking suite [88]. This suite contains a wide variety of complex real-world programs, including a Perl interpreter, XML parser and simulations. Additionally, we evaluated Delta Pointers on Nginx 1.10.2. We used Intel Xeon E5-2630v3 machines with 16 cores at 2.40 GHz and 64 GB of memory, running 64-bit CentOS 7.2.1511. Each overhead number is the median of 16 iterations of the same program (using the reference workset for SPEC), and we manually verified standard deviations to be negligible. For the baseline, we enabled link-time optimizations. Two of the 19 benchmarks are not compatible out-of-the-box with Delta Pointers: *403.gcc* uses upper pointer bits of pointers larger than 32 bits for page ordering, and corrupts a pointer tag by using NULL pointer subtraction and addition instead of a regular typecasts for type conversion between pointers and integers. *450.soplex* is incompatible with per-pointer bounds checkers: it patches pointers to point to out-of-bounds locations using pointer subtraction after reallocating a buffer containing pointers. This violation of the C standard is well-documented in related work [113, 158], and other researchers either patch or omit these programs as well. In order to avoid the impact of omitting benchmarks on the overall overhead, we wrote small source patches for *403.gcc* to configure pointer bitsize at 32 bits and to preserve a pointer tag at a single NULL pointer subtraction, and for *450.soplex* to fix the delta tag on patched pointers after `realloc`. The evaluation numbers thus include the entire C/C++ subset of SPEC CPU2006 (19 benchmarks in total).

We have benchmarked Delta Pointers both with and without the optimizations from Section 3.6.4. Figure 3.4 shows the measured runtime overhead for these configurations (raw runtimes are in Table 3.2 at the end of this chapter). In all cases, Delta Pointers show a negligible memory overhead. The instrumentation causes 41% geometric mean (geomean) runtime overhead, which is reduced to 35% by optimizations. In comparison, our implementation of using branches

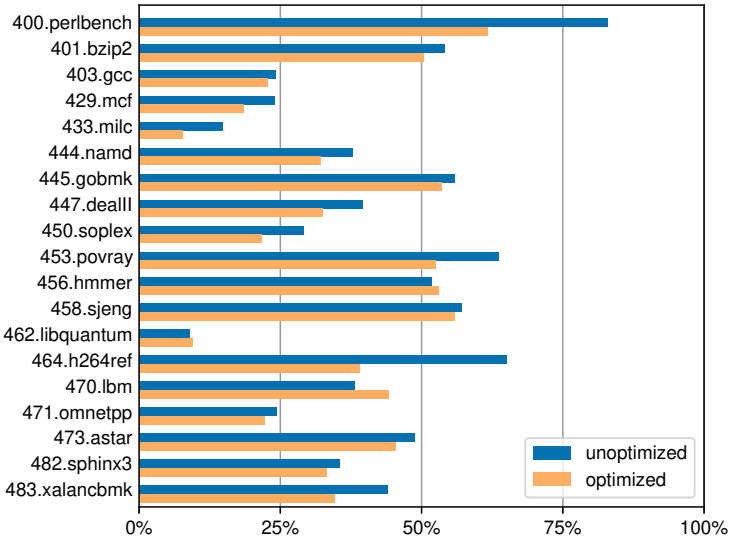


Figure 3.4: Runtime overhead of SPEC CPU2006 for our Delta Pointers prototype.

for upper-bound checks (which is equivalent to SGXBounds without underflow checks) achieves 48%, confirming that bitwise and arithmetic instrumentation is significantly faster than using branches. The instrumented binaries are on average 80% bigger, both in terms of instructions and file size.

To accurately compare these results with related work, we tried to reproduce competing results on the same setup in order to compare to the same efficient baseline on modern hardware. We evaluated SGXBounds on our setup and obtained 94% geomean overhead which is much higher than the 55% reported in the paper. After consistently seeing these results across machines, we contacted the authors but together we were not able to determine the root cause of this difference. The authors of Low-Fat Pointers were unable to share their prototype due to it being in an alpha state, and Baggy Bounds [6] is closed-source. We could easily evaluate AddressSanitizer because it is part of LLVM. The run times obtained on our experimental setup are in in Table 3.3 at the end of this chapter.

Overall, we can see that our overflow detection design is far more efficient than ASan, which has 80% overhead. We only have slightly lower compatibility: we require small source patches for two SPEC 2006 benchmarks whereas ASan requires this for one benchmark. ASan also has a large memory overhead, whereas Delta Pointers have negligible memory overhead. Also note that ASan has weaker spatial detection guarantees, since it can only detect contiguous overflows. On the subset of SPEC 2006 benchmarks supported by SGXBounds, Delta Pointers have a 35% overall overhead, compared to the 94% of SGXBounds, while achieving higher compatibility (SGXBounds cannot run 6 benchmarks because it does not deal with

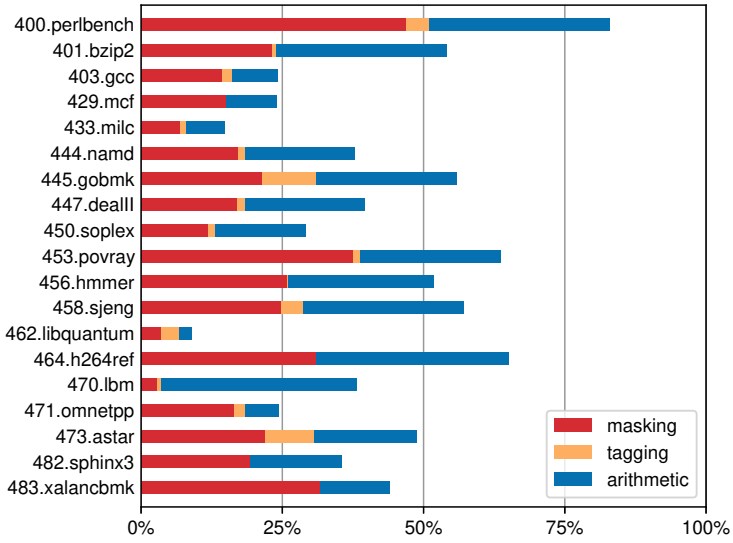


Figure 3.5: Runtime overhead of SPEC CPU2006 for different components of Delta Pointers instrumentation (with optimizations disabled).

some issues described in Section 3.5). Low-Fat Pointers does not support 4 out of the 19 SPEC benchmarks, but can be benchmarked using manual source fixes. The paper [64] reports 64% overhead[3] versus 35% for Delta Pointers. In addition, Low-Fat does not cover globals or the `NULL` pointer and provides lower compatibility overall.

To gain insight in the origin of the measured overhead, we have independently measured the overheads of different instrumentation components (with optimizations disabled). The results are in Figure 3.5. We observe that the overhead of masking at every pointer dereference is geomean 20%. This is perhaps higher than expected since the operations are bitwise ANDs which are expected to be fast. But the performance impact of register pinning and pipeline stalls, as described in Section 3.5.4, evidently proves non-trivial and likely requires hardware optimizations to further reduce the overhead. Tagging all allocations with delta tags is cheap, adding only 2 percent point overhead. This is expected, since allocations typically happen outside of the main computation loops. Finally, like masking, instrumentation on pointer arithmetic is higher than expected at 19 percent point, making the overall overhead 41%. We have investigated the assembly generated for our instrumentation by the compiler and concluded that the instrumentation hinders optimizations made by the LLVM x86-64 backend. In particular, the instruction selec-

[3] 64% overhead is reported for the latest Low-Fat Pointers version which includes protection of global variables, and is higher than the 54% for only stack+heap reported in the published Delta Pointers paper.

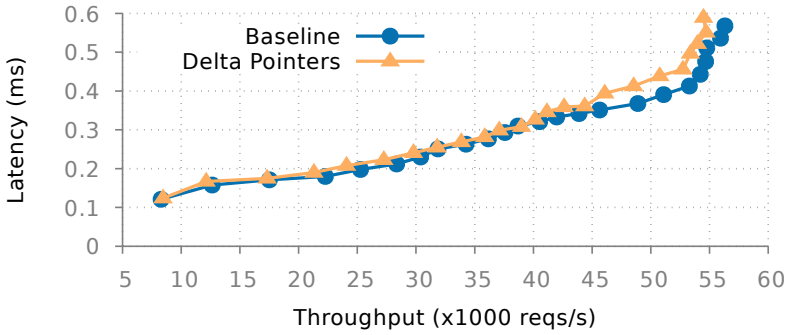


Figure 3.6: Overhead of Nginx web server for Delta Pointers

tion chooses to emit separate addition and multiplication instructions for pointer offset computations rather than using efficient scaling-based addressing mode as supported by `lea` and `mov`. This is especially the case for dynamic offsets that cannot be constant-folded. Unfortunately, dynamic offsets are prevalent in SPEC: we found that 72% of all pointer arithmetic instructions in the reported benchmarks use dynamic offsets. Thus, although our current overhead is already competitive, (admittedly non-trivial—due to the LLVM architecture) optimizations of the instrumentation of dynamic pointer arithmetic in the compiler backend may improve the results even more.

We have also tested Delta Pointers on the Nginx web server. The performance of Nginx primarily depends on I/O, and Nginx does relatively few pointer operations. Because of this we observed negligible effects on performance, as shown in Figure 3.6. Our benchmarks were performed requesting 64 byte pages with 8 workers over a 54 Gbit/s link. On average we observe a 4% increase in latency, with a maximum of 6% for the unsaturated case going down to 3% when saturation is reached. When the server is saturated we observe only a 2% drop in throughput for Delta Pointers.

3.7.2 Security

We have evaluated the effectiveness of Delta Pointers by examining a number of recent common vulnerabilities and exposures (CVEs) [143]. To be able to compare to previous work, we have examined all CVEs reported in recent work [66, 113]: 8 CVEs across 6 popular programs, including Heartbleed in OpenSSL and bugs in Nginx and PHP. Rather than only checking if existing exploits are circumvented, we used manual analysis to determine if the *exploitability* of an attack is affected.

As detailed below, Delta Pointers completely prevent 7 of the 8 analyzed attacks, with the uncaught bug only supported by a single existing bounds checker,

demonstrating that Delta Pointers offer practical security guarantees. The undetected bug is not fundamental to the design of Delta Pointers, but is outside the protection boundary of our implementation. Adding support for the bug is trivial (a single line of code) by treating `recv` as a memory intrinsic. This could be done for all standard C library functions, as done by SGXBounds, to extend protection to outside the protection boundary.

CVE-2011-4971 in MemCached 1.4.15 A signed integer is set to a negative value by the attacker, and passed as a large unsigned size to `memcpy`. Our instrumentation on memory intrinsics covers this case, preventing out-of-bounds reads.

CVE-2013-2028 in Nginx 1.4.0 An attacker-controlled negative signed integer is passed as a large unsigned size to `recv` which copies data into a limited-size buffer. Delta Pointers cannot not detect this case since the write resides in an uninstrumented libc function. To cover this vulnerability, functions like `recv` could be encapsulated in wrappers that implement checks on the underlying memory accesses. Note that this requires manual analysis of the semantics of all library functions that access memory buffers based on parameter values which is only done by SGXBounds, so the bug is not covered automatically by any other existing defense.

CVE-2014-0160 in OpenSSL 1.0.1f (Heartbleed) A 2-byte attacker controlled response length is used to transmit back a buffer of an attacker-controlled size, allowing the attacker to leak up to 64KB of memory, including private keys. The buffer pointer is correctly tagged and the overread is detected successfully.

CVE-2016-1234 in glibc-2.19 While glibc is not compatible with LLVM out-of-the-box, we still analyze this CVE to compare effectiveness of our design to that of Low-Fat Pointers which reports it. The bug is a stack buffer overflow due to an access with the length of a directory name as offset, which can be up to 510 bytes larger than the buffer size on for instance the NTFS file system. Our Delta Pointers design would correctly tag the allocated buffer and prevent the attack.

CVE-2016-2554 in PHP-5.5.31 A string inside `struct _tar_header` is assumed to be null-terminated but can be attacker controller. By crafting a struct without any null-bytes this intra-struct overflow can be extended far beyond the size of the struct. This finally ends up in a `strncpy` with the overflowed size of the struct, which Delta Pointers detect because of our `strncpy` instrumentation.

CVE-2016-3191 in PCRE2-10.20 An attacker-controlled regex can cause a contiguous overflow on a stack buffer, as the `(*ACCEPT)` verb will write a closing parenthesis for any currently open parenthesis, without checking for the presence of such

closing parenthesis nor whether there is space in the buffer. Our Delta Pointers design easily detects this case.

CVE-2016-6289 in PHP-7.0.3 Similar to CVE-2011-4971 a signed integer is passed to a `memcpy` call. Our Delta Pointers implementation instruments the intrinsic and detects any overflow.

CVE-2016-6297 in PHP-7.0.3 An attacker can supply a large string triggering an integer overflow in `strlen`. The resulting length is then passed to `memcpy` where it is cast to a `size_t` similar to CVE-2011-4971, which is detected.

SPEC CPU2006 We have also confirmed a number of benign buffer overflows in SPEC CPU2006 which are reported by related work [113, 186]. `perlbench` contains a benign buffer overread in a `memcmp` call, which is caught by Delta Pointers because of our intrinsic handling. `h264ref` contains two bugs: one involving a global variable and one on the stack. The stack-based overflow is entirely optimized away by LLVM during vectorization (as LLVM can statically determine there is undefined behavior). If we disable these optimizations, Delta Pointers detect the bug. The bug where a global variable is overflown is also detected by Delta Pointers, in contrast to, for instance, Low-Fat Pointers. For the performance evaluation of Delta Pointers, we fixed these bugs using the source patch from AddressSanitizer.

3.8 Discussion

Bounds narrowing Our prototype does currently not support *bounds narrowing*, where bounds of sub-objects in composite types are enforced (e.g., an array in a struct). Since our Delta Pointers design records per-pointer metadata, such a feature could easily be added. Such strict enforcement of bounds is knows to cause compatibility problems [46, 158].

Integer overflow on pointers Most processor architectures implement arithmetic overflow on regular integer additions: when all bits in an integer are set and 1 is added, the number wraps around to zero. In our pointer encoding scheme, an attacker may be able to clear the overflow bit by adding a very large offset that causes such an overflow. This can normally be mitigated by simply limiting offsets to 32 bits (which is already normally the case in real-world programs), but, in some cases, the attacker might be able to use a pointer addition inside a loop to iteratively overflow the pointer. Thus, in order to provide complete protection on the upper bound, Delta Pointers require an upper bound on the result of pointer additions. This is called *saturation arithmetic*. Saturating operations *clamp* their results to a given minimum and maximum, typically with all-zero and all-one bits respectively.

Some architectures have dedicated instructions for saturation arithmetic. For instance, ARM offers the `UQADD` instruction to perform saturating addition on 64-bit unsigned integers, which constitute our use case. Intel x86-64, however, only supports 8/16-bit saturation through `PADDUSB`/`PADDUSW`. 64-bit saturation on x86-64 is most efficiently performed by conditional (but non-branching) instructions (`CMOVcc`) which replace the result of an addition based on the value of the `FLAGS` register. To support saturation arithmetic, Delta Pointers could optionally use conditional instructions to replace the result of pointer arithmetic with the maximum integer value. Such instructions are reasonably fast since they operate only on registers, but they must be inserted on all pointer arithmetic, hence resulting in higher overhead. For our Delta Pointers prototype, we felt this was not a feature to prioritize given the additional performance cost and limited security improvement. Namely, the feature is only necessary if an attacker can add an offset of `(1 << 31) + object_size` bits to a pointer, either at once or iteratively, without dereferencing it in the meantime (since the intermediate pointers are detected as out-of-bounds). This is rarely an option in practice, since a pointer computed inside a loop is usually dereferenced inside the loop, and otherwise hoisted out of the loop by compiler optimizations. We have manually confirmed this for all the vulnerabilities analyzed in Section 3.7, for which saturation arithmetic would thus not provide any security improvement.

Unaligned access When a pointer is cast to an arbitrary type and dereferenced, the number of dereferenced bytes may differ from the allocated pointer type. Delta Pointers do not support detection of such *unaligned* accesses, since the situation only arises in the context of a type confusion bug which is not in our threat model. For intellectual curiosity, we have, however, implemented optional support for unaligned access detection in the form of an additional pointer arithmetic that adds the dereferenced number of bytes minus one to the delta tag before each dereferencing instruction. This adds 3% overhead on SPEC 2006.

Delta tag compression Delta tags take up half the pointer in the current design of Delta Pointers, severely limiting the address space (and thus ASLR entropy) in return for strong memory safety guarantees. Reducing the number of bits needed for the delta tag could alleviate this trade-off, as done in similar schemes. For example, Baggy Bounds [6] compresses object size tags in pointers by allocating power-of-two sized objects, storing only the exponent. Compression for Delta Pointers is not trivial since the delta tag stores the distance to the end of the object rather than the object size. At first sight it might seem possible to compress this by aligning all objects and their size to a number of *compression bits* (n). However, this naive scheme breaks when a pointer is unaligned with respect to its compression. For example, if we align all objects to 8 bytes ($n = 3$), we store 3 fewer bits in the

tag. This makes it impossible to update the tag with offsets smaller than 8 bytes, e.g., `p=(char*)malloc(N)+1`. When done iteratively, 8 such pointer additions would overflow the object without being detected.

Arbitrary compression is possible by adding additional arithmetic operations to each pointer modification. The intuition is that, when aligning all objects to n bits, the lower n bits of the delta tag and the address contain the same information. Thus, we can store this information in only the address itself, enabling compression of the tag. Any addition smaller than 2^n only occurs on the address, but if it carries into the $(n+1)$th address bit we continue the addition on the delta tag. The remainder (the part of the offset that is a multiple of 2^n) is added directly to the tag. The following code shows how this is done using bitwise operators, when adding offset a to a pointer while using n compression bits:

```
carry   = ((ptr_old ^ ptr_new ^ a) >> n) & 1
tag_new = tag_old + (a >> n) + carry
```

The first line determines whether the lower n bits of the address carried into the next bit, and the second line replicates the carry on the tag. This scheme offers a larger address space, but incurs memory overhead due to alignment, and runtime overhead due to the additional instrumentation required.

Backend optimization Section 3.7 details the performance hit of pointer arithmetic instrumentation on x86-64. Future work could feature an optimization of the LLVM backend that allows for efficient code generation of instrumented pointer arithmetic, better utilizing the scaling-based addressing mode of x86-64 instructions.

3.9 Related work

Overflow detection There has been a plethora of research on buffer overflow detection over the past decades. Table 3.1 present a summary of the major systems. Early systems often relied on fat pointers which had a high overhead and introduced many compatibility issues, often requiring programmers to change existing code [10, 99, 151]. The system proposed by **Jones and Kelly** [100] instead relied on external metadata, associated *per-object*. Since such per-object designs retrieve pointer bounds by looking at the objects, they cannot support temporarily out-of-bound pointers, and perform their checks during arithmetic. Later designs attempted to fix these limitations [178] and performance issues [61].

The first practical design was that of **Baggy Bounds** [6], which encodes metadata in the memory layout. Similarly to Delta Pointers, Baggy Bounds also uses pointer tagging to store the size-class of an object, and encodes encodes out-of-bound pointers as invalid (non-canonical) to cause automatic hardware crashes. Instead of masking pointers before every memory access, Baggy Bounds instead

System	C++	Metadata	Checks	Passing OoB ptrs	Non-linear	Benchmarks	+Runtime	+Memory
Softbound [150]	✗	Table	Deref	✓	✓	Random[a]	67%	64%
Baggy Bounds [6]	✗	Layout	Arith	✓[b]	✓	SPEC2000 - 3	72%	11%
PAriCheck [221]	✗	Shadow	Arith	✓	✓[c]	SPEC2000 - 4	96%	18%
LBC [87]	✗	Shadow	Deref	✓	✗	SPEC2000 - 2	22%	7.7%
ASan [186]	✓	Shadow	Deref	✓	✗	SPEC2006	80%	237%
Intel MPX [158]	✓	Table	Deref	✓	✓	SPEC2006	139%	90%
LowFat [65, 66]	✓	Layout	Deref	✗	✓	SPEC2006	64%	5.2%
SGXBounds [113]	✓	Tag	Deref	✓	✓	SPEC2006 - 6	89%	0.1%
Delta Pointers	✓	Tag	—	✓	✓	SPEC2006	35%	0%

[a] On an arbitrary subset of SPEC '95, 2000, 2006 and Olden.
[b] Only up to $alloc\_size/2$ on 32-bit.
[c] Unless wrap-around on 16-bit labels occurs.

Table 3.1: Comparison of overflow checkers, including our own system Delta Pointers. Most evaluations have (nonoverlapping) sets of benchmarks for the results, making the overhead numbers difficult to compare.

places tags for valid pointers in the upper bits of the lower 48 bits, and creates aliased mappings for each tag. This way every tagged pointer is still valid in the address space, with the limitation that even fewer bits are available for both the tag and the pointer. **Low-Fat pointers** [65, 66] takes the baggy bounds design and optimizes it even more by grouping size-classes together in the address space. By sacrificing compatibility of not supporting out-of-bound pointers between contexts Low-Fat can achieve a higher performance. Like any other bounds checker, the Delta Pointers design supports out-of-bound pointers.

Instead of relying on per-object metadata, Delta Pointers rely on *per-pointer* metadata. **SoftBound** [150] transforms the source to keep track of the metadata for pointers, and stores them separately whenever pointers leak to memory. Because of its limited static analysis, SoftBound suffers from a low compatibility. **Intel MPX** [158] achieves a similar design with the support of hardware, introduced with the Skylake microarchitecture but incurs high overhead.

SGXBounds [113] presents a design well geared towards the current version of SGX, which has a limited address space. It encodes the upper bound inside the pointers, with a design based on **Boundless** [32]. Boundless stores distance information similar to Delta Pointers, but only to ultimately compute the upper bound as used by SGXBounds. These systems thus still require branches, and rely on memory accesses to record lower bounds. SGXBounds suffers from a low compatibility due to its limited static analysis, a problem which Delta Pointers address. Delta Pointers could also work well inside SGX enclaves. Outside of enclaves SGXBounds is shown to suffer from much higher overheads.

Some alternative designs do not record the bounds information itself. **PAriCheck** [221] instead tags every few bytes with a label, and during pointer arithmetic enforces that every pointer points to memory with the same tag. Sadly such a design suffers from impractical high overheads. **LBC** [87] and **AddressSanitizer** [186] instead place guard zones around every object, and verify every memory access is outside of a guard zone. Guard zones and their metadata incur a large memory overhead, and moreover, can only detect contiguous buffer overflows.

Pointer tagging The concept of tagged pointers has been used for decades, but generally concerns the lower bit(s) of a pointer [9, 72]. Baggy Bounds [6], Boundless [32], SGXBounds [113], and Mid-Fat [110] all store tags in the upper bits of pointers. Our design provides a more comprehensive analysis increasing compatibility of pointer tagging over these approaches. Hardware support for pointer tagging can be found in recent architectures with ARMv8's virtual address tagging [7] and Oracle's SPARC-M7 SSM/VA masking [160].

3.10 Conclusion

In this chapter we presented Delta Pointers, a design for a fast and compatible buffer overflow detector. In contrast to existing solutions, Delta Pointers do not rely on memory lookups nor branches, yielding a competitive, low-overhead design with 35% performance overhead and negligible memory overhead. Our design relies on pointer tagging to maintain the distance from the current pointer to the end of the object to implicitly invalidate overflowed pointers. We hope our findings on pointer tagging will encourage future research, and as such, our framework and Delta Pointers prototype are available open source.

Contribution

This chapter was previous published as a research paper in collaboration with other authors, and for this paper I share first authorship with Taddeüs Kroes. We wrote the prototype and paper together in a join effort, with the following exceptions: I wrote the `libshrink` library to shrink a program's address space, and Taddeüs implemented all optimizations to elide instrumentation on safe pointers and was responsible for the open source release of Delta Pointers.

Benchmark	Lang.	Components		Delta Pointers		Branches ubound		Related work	
		Mask	Mask + tag	No opts	Opts	No opts	Opts	ASan	SGXBounds
400.perlbench	C	1.47	1.51	1.83	1.62	3.34	1.97	4.01	-
401.bzip2	C	1.23	1.24	1.54	1.50	1.72	1.63	1.70	2.22
403.gcc	C	1.14	1.16	1.24	1.23	1.47	1.34	2.23	-
429.mcf	C	1.15	1.15	1.24	1.18	1.36	1.29	1.70	1.58
433.milc	C	1.07	1.08	1.15	1.08	1.27	1.09	1.35	1.59
444.namd	C++	1.17	1.18	1.38	1.32	1.42	1.36	1.52	1.84
445.gobmk	C	1.21	1.31	1.56	1.53	2.17	1.91	1.66	2.18
447.dealII	C++	1.17	1.18	1.40	1.32	1.56	1.32	2.26	-
450.soplex	C++	1.12	1.13	1.29	1.22	1.37	1.22	1.57	-
453.povray	C++	1.37	1.39	1.64	1.52	2.47	2.00	2.72	-
456.hmmer	C	1.26	1.26	1.52	1.53	2.00	1.60	1.91	2.51
458.sjeng	C	1.25	1.29	1.57	1.56	2.47	1.98	1.73	2.32
462.libquantum	C	1.03	1.07	1.09	1.09	1.12	1.11	1.08	1.19
464.h264ref	C	1.31	1.30	1.65	1.39	2.32	1.44	2.15	-
470.lbm	C	1.03	1.03	1.38	1.44	1.21	1.19	1.01	1.63
471.omnetpp	C++	1.16	1.18	1.24	1.22	1.43	1.30	2.09	-
473.astar	C++	1.22	1.30	1.49	1.45	1.76	1.57	1.54	1.84
482.sphinx3	C	1.19	1.17	1.35	1.33	1.74	1.57	1.69	2.24
483.xalancbmk	C++	1.32	1.30	1.44	1.35	1.84	1.64	2.08	2.68
Geomean		1.20	1.22	1.41	1.35	1.72	1.48	1.80	-
Subset sgxbounds		1.17	1.19	1.38	1.35	1.63	1.47	1.55	1.94

Table 3.2: Normalized runtime overheads on SPEC CPU2006 for Delta Pointers, an implemention using the pointer tagging framework on Delta Pointers but using branches for upper bound checks ("Branches ubound"), and related work.

Delta Pointers

	Baselines			Components		Delta Pointers		Branches ubound		Related work	
Benchmark	LTO	ASan[a]	SGXBounds[b]	Mask	Mask + tag	No opts	Opts	No opts	Opts	ASan	SGXBounds
400.perlbench	262	282	-	386	396	480	424	878	518	1133	-
401.bzip2	440	432	438	542	545	678	662	758	719	736	974
403.gcc	255	259	-	291	296	317	313	374	342	577	-
429.mcf	228	227	227	262	262	283	270	310	295	387	359
433.milc	458	482	491	490	494	525	493	581	501	650	780
444.namd	331	328	328	388	392	457	438	471	451	498	604
445.gobmk	380	404	389	462	498	593	584	827	727	671	850
447.dealII	225	253	-	263	266	315	298	350	297	572	-
450.soplex	197	195	-	220	222	254	239	270	240	306	-
453.povray	118	130	-	163	164	194	180	293	236	356	-
456.hmmer	380	382	385	478	479	577	582	762	610	731	966
458.sjeng	417	427	413	520	536	655	650	1032	828	737	957
462.libquantum	336	339	343	347	358	366	368	375	374	367	409
464.h264ref	468	481	-	612	610	774	651	1085	674	1034	-
470.lbm	353	354	339	362	365	487	509	427	421	357	551
471.omnetpp	280	291	-	326	331	348	342	400	365	607	-
473.astar	319	351	343	388	416	474	463	561	500	542	630
482.sphinx3	453	428	446	540	529	614	604	787	712	724	1001
483.xalancbmk	171	184	179	225	223	246	230	314	280	383	480

[a] No LTO, since ASan does not support LTO on all benchmarks.
[b] Linked against musl, no LTO but all bitcode is combined in a single module with O3, libc++ inlined as bitcode.

Table 3.3: Raw runtimes of SPEC CPU2006 in seconds, corresponding to the overheads in Table 3.2. All numbers were gathered on the DAS-5 cluster [210]. Each number is the median of 16 runs. Note that Delta Pointers and other overflow checkers each require a different build configuration and baseline.

4 Secure and Efficient Multi-variant Execution Using Hardware-assisted Process Virtualization

Memory error exploits rank among the most serious security threats. Of the plethora of memory error containment solutions proposed over the years, most have proven to be too weak in practice. Multi-Variant eXecution (MVX) solutions can potentially detect arbitrary memory error exploits via divergent behavior observed in diversified program variants running in parallel. However, none have found practical applicability in security due to their non-trivial performance limitations.

In this chapter, we present MvArmor, an MVX system that uses hardware-assisted process virtualization to monitor variants for divergent behavior in an efficient yet secure way. To provide comprehensive protection against memory error exploits, MvArmor relies on a new MVX-aware variant generation strategy. The system supports user-configurable security policies to tune the performance-security trade-off. Our analysis shows that MvArmor can counter many classes of modern attacks at the cost of modest performance overhead, even with conservative detection policies.

4.1 Introduction

For more than a quarter of a century and despite a plethora of proposed solutions, memory errors in C and C++ programs still rank among the most serious security concerns today [195, 204]. Even an unsophisticated memory error exploit like Heartbleed can easily compromise the private data of countless users worldwide with serious consequences [67].

Modern operating systems deploy several measures to protect against memory error exploits, but all of them can be circumvented with varying amounts of effort. For example, widely deployed security defenses such as data execution prevention (DEP) [137], address space layout randomization (ASLR) [169], and stack canaries [50] can all be bypassed by modern code-reuse attacks [25, 185]. Stronger security defenses proposed by the research community, either require recompilation of the program and all shared libraries [6, 61, 149, 150] (limiting deployability), or protect only against a subset of all possible memory attacks (limiting security). As an example, popular control-flow integrity (CFI) solutions that protect against control-flow diversion attacks [2, 170, 198, 205, 224] are ineffective against data-only attacks (such as Heartbleed) and possibly even against control-flow diversion attacks that piggyback on legal control flows in the program [36, 37, 58, 79].

The need for defenses that protect against arbitrary attacks has led to a scramble for more comprehensive solutions—most notably Multi-Variant eXecution (MVX)[1] [52, 91]. MVX systems, first proposed by Cox et al. [52] in 2006, run two or more memory-diversified but semantically equivalent software variants in parallel and detect memory attacks from semantically divergent behavior. These variants run on the same machine (utilizing many-core CPUs) and synchronize at the system call (syscall) level. While such systems have been around for nearly a decade, the run-time performance of traditional MVX implementations [20, 33, 180, 209] is so poor—due to their costly syscall monitoring mechanisms—to make them unusable in practice. Also, the limited variant generation strategies in existing solutions often do not offer adequate protection against more sophisticated memory attacks. Recent MVX efforts have therefore focused either on generating better variants so as to detect (some) modern attacks but with no improvement in performance [180, 209], or on improving the performance by efficient in-process implementations which are, unfortunately, not suitable for security enforcement purposes and target reliability instead [91].

In this chapter, we present MvArmor, an MVX system which relies on a new secure and efficient multi-variant design to counter arbitrary memory error exploits. Our design leverages hardware-assisted process virtualization to place the application-level MVX monitor directly in the syscall path of each of the running variants. This approach is efficient, as it does not incur the frequent context

[1] Also known as N-variant or dual execution and closely related to N-version execution.

switches from/to external monitoring processes required by traditional process tracing-based MVX implementations [33, 180, 209]. Furthermore, given that the process virtualization layer can grant the MVX monitor access to privileged CPU features [18], our design is particularly amenable to optimizations [19, 172]. At the same time, this approach is secure, as it relies on hardware-enforced protection rings to completely isolate the execution of the MVX monitor by construction—unlike prior in-process implementations [91]—protecting it from known and unknown attacks. Furthermore, unlike prior implementations running entirely in the kernel [52], our design separates the application-level MVX monitor from the rest of the system, limiting the trusted computing base (TCB).

To counter arbitrary attacks effectively, we complement our MVX design with a new MVX-aware variant generation strategy, which seeks to provide strong security and performance guarantees with no manual effort. Our strategy relies on per-variant allocator abstractions to carefully and efficiently control the memory layout across the running variants. This strategy provides deterministic security guarantees against arbitrary memory error exploits when possible—or strong probabilistic guarantees otherwise. Finally, MvArmor supports flexible security policies tailored to different classes of modern attacks (e.g., arbitrary code execution or information disclosure), allowing users to tune the performance-security trade-off according to their needs.

Summarizing, our contributions are:

- We propose an MVX design based on hardware-assisted process virtualization. Our design efficiently separates the execution of the MVX monitor from both the running variants and the underlying kernel, providing a superior performance and security design point compared to prior efforts.

- We propose a novel variant generation strategy based on MVX-aware allocator abstractions. Our strategy is efficient and, when used in combination within our MVX design, provides strong security guarantees against both traditional and modern memory error exploits.

- We present MvArmor, a secure and efficient MVX system. MvArmor implements our design on top of Dune [18] to protect commodity Linux programs and offers flexible security policies to encourage deployment. We evaluate MvArmor with standard benchmarks and popular real-world server programs and show that MvArmor provides a powerful defense against arbitrary memory attacks with much better performance than any existing security-related MVX solution (9% overhead on SPEC CINT2006 and just 55% on average for server applications even with the most conservative security policy).

4.2 Background

Every MVX system contains two major components: a monitor which runs and synchronizes the variants and a variant generation strategy. Both have a strong impact on the security and performance of the overall system and have been the focus of extensive research in the past decade.

4.2.1 Monitor

The MVX monitor is responsible for comparing and synchronizing the execution of the running process variants. These variants all run on the same system, and ideally each have a numbers of cores dedicated to them—we assume that a number of cores can be explicitly dedicated to particularly security-sensitive applications in modern many-core architectures. The monitor itself might consist of several processes, for instance one per variant, communicating via shared memory. The entire MVX system (including the monitor) is designed to be application- and user-transparent. For example, in the case of a web server running under MVX, the user's request will get distributed to all variants. The monitor will also combine the responses of all web server variants and give the user the illusion she is directly talking to a single web server instance. Furthermore, whenever these responses (or other operations) are not equivalent across variants, the monitor can immediately detect an attack attempt (as normal operations should never trigger divergent behavior) and stop the variants before the attacker could do any harm. In general, MVX does not lead to more filesystem and socket I/O, as the monitor effectively executes all syscalls, not every variant. On the other hand, all variants will have to execute all instructions and memory reads/writes by themselves, leading to more overall CPU usage and potential memory bandwidth issues.

In most cases, syscalls are used as synchronization points, as they are generally the primary way for each process to interact with the environment (e.g., file system operations or socket operations). A monitor operating at the syscall level can capture and control external behavior while still allowing for individual variants to exhibit different internal behavior.

A monitor must be able to intercept syscalls and their arguments to compare process behavior across variants. It must also be able to rewrite arguments, block syscalls, and modify the return value (or memory) to ensure uniform and side effect-free syscall handling across all the variants. In general, the monitor needs to ensure all the variants are exposed to the same environment view and information (e.g., PIDs) to avoid unintentionally divergent behavior.

Several strategies have been used in prior MVX systems to intercept syscalls, but all of them suffer from important performance and/or security limitations. To gather insights into these limitations, we evaluated the run-time overhead induced

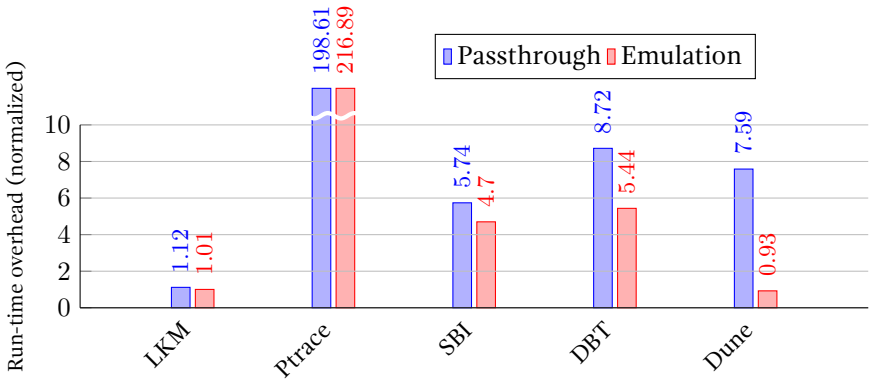


Figure 4.1: Overhead induced by several syscall interposition strategies (passthrough and emulation mode) for a microbenchmark repeatedly issuing `getpid` syscalls.

by existing syscall interposition strategies for a simple microbenchmark repeatedly issuing `getpid` syscalls. Figure 4.1 presents our results in both *passthrough* (i.e., forwarding the original syscall to the underlying OS kernel) and *emulation* (i.e., immediately returning the result to the application) mode.

As shown in the figure, an MVX monitor based on a loadable kernel module (LKM) implements by far the most efficient syscall interposition strategy in both modes of operation, as it does not introduce additional context switches and can directly access the process state. The problem with this strategy, adopted by early MVX systems [52], is that the monitor runs entirely in the kernel. This results in a substantial increase of the trusted computing base (TCB) and poor deployability: a single bug in the monitor could affect the entire system, and the internal kernel API is very volatile.

At the opposite side of the spectrum lie traditional `ptrace`-based MVX implementations [33, 90, 133, 180, 209], which rely on the UNIX process tracing API to implement a deployable but highly inefficient syscall interposition strategy. This strategy introduces multiple context switches between the monitor and the traced process per syscall, resulting in by far the highest overheads (up to ∼217 times) for our microbenchmark. On 64-bit systems a monitor using `ptrace` cannot block syscalls, making emulation even more expensive than passthrough. Syscall monitoring using `ptrace` is also susceptible to TOCTOU attacks [177], which are hard to resolve due to the large latency between operations and limited access between the monitor's and the application's address spaces.

More recent MVX monitor implementations rely on static binary instrumentation (SBI) to rewrite the binary and replace any syscall instruction (e.g., `int $0x80` or `syscall`) with a call into the monitor. As shown in Figure 4.1, this syscall interposition strategy is much more efficient (up to ∼5 times overhead, based on the

Dyninst SBI framework [34]), as it requires no context switches and even no mode switches in emulation mode. Unfortunately, this strategy is not generally suitable for security applications, as an attacker can tamper with the in-process monitor state or simply run uninstrumented unaligned syscall instructions (not part of the normal instruction stream, as x86 does not enforce alignment) to bypass syscall interposition and evade detection altogether. An implementation based on dynamic binary translation (DBT), in turn, would be able to instrument both aligned and unaligned instructions and solve the latter problem, at the cost of slightly higher syscall interposition overhead (up to ∼9 times overhead in our experiment, based on the DynamoRIO DBT framework [31]), but also a non-trivial impact during syscall-free execution. To fully address the former problem, a DBT-based solution needs to deploy additional instrumentation [170] (e.g., software-based fault isolation [211]), but this would also further increase the overhead during syscall-free execution.

MvArmor, instead, relies on hardware-level process virtualization to implement syscall interposition. For this purpose, we use Dune, which virtualizes regular Linux processes and places them in their own (hardware-supported) virtual environment. As shown in Figure 4.1, this strategy is very efficient compared to other techniques (up to ∼7 times overhead on `getpid()`) and meets all our security demands: small systemwide TCB, fully isolated monitor with no in-process state, non-bypassable (trap-based) syscall interposition mechanism. Furthermore, the excellent performance in emulation mode and the ability to access privileged CPU features provide interesting opportunities for libOS-style optimizations [19, 172].

4.2.2 Variant generation

The variant generation strategy has a strong impact on the classes of attacks addressed by the resulting MVX system. Ideally, any attack should eventually result in divergent behavior among variants. For instance, relying on ASLR for variant generation makes it unlikely that two or more variants share code pages at the same addresses, making code-reuse attacks such as ROP [188] more difficult. By extending this strategy to use MVX-aware (i.e. non-overlapping) address spaces, traditional code-reuse attacks can be fully prevented. This is because no code pointer can ever be valid in more than one variant, and thus will cause a fault while dereferencing it in all but one of the variants [20, 33, 52]. While this approach will also stop arbitrary memory read and write attacks that rely on absolute memory object locations (e.g., data pointer overwrites), it cannot stop attacks that only rely on the relative distance between memory objects. For instance, stack- or heap-based buffer overreads that disclose private data (e.g., cryptographic keys) [67] and overflows that corrupt sensitive non-pointer data (e.g., UIDs) [41] will still

work reliably, as these attacks only use relative memory accesses.

Alternative variant generation strategies include reversing the direction of the stack [180] and randomizing the instruction set [52]. These strategies add little additional security (e.g., addressing only stack-based and code injection attacks, respectively) and often introduce non-trivial overhead by themselves.

MvArmor, in contrast, relies on a new MVX-aware variant generation strategy, which seeks to minimize the run-time performance impact while providing strong security guarantees against arbitrary classes of attacks that rely on both absolute and relative accesses in memory. Security policies allow the user to choose between increasing levels of protection (both probabilistic and deterministic), at the cost of a larger performance overhead.

Finally, the advantage of using an MVX system over other approaches with the same goals is that it can simultaneously protect against multiple of these types of attacks with less overhead (given enough spare CPU cores and resources), and does not generally require access to the source or recompilation of system libraries [149, 150].

4.3 Threat model

We assume a strong threat model where an attacker can interact with the target program repeatedly, exploiting vulnerabilities to read or write arbitrary data from/to memory. In particular, we assume an attacker can rely on both *relative* (e.g., buffer overread/overflow and partial pointer overwrites) and *absolute* arbitrary memory read/write primitives (e.g., pointer overwrites). We also assume both *spatial* (e.g., buffer overflows) and *temporal* (e.g., use-after-free) memory attacks [21]. Based on these primitives, we assume an attacker may pursue any of the following goals (in line with the characteristics of modern attacks [36, 92]):

- **Arbitrary code execution**: An attacker could execute arbitrary code, for example issuing an `execve` syscall using ROP [188] or other code-reuse techniques [183].
- **Information disclosure**: An attacker could leak sensitive data from the target program, for example cryptographic keys as in the case of Heartbleed [67].
- **Information tampering**: An attacker could tamper with sensitive data, for example UIDs to escalate privileges, or mount other non-control-data attacks [41, 92].

4.4 Overview

Figure 4.2 shows the main components of MvArmor, which all run inside a virtualized environment [18]. Additionally, MvArmor can provide the same functionality

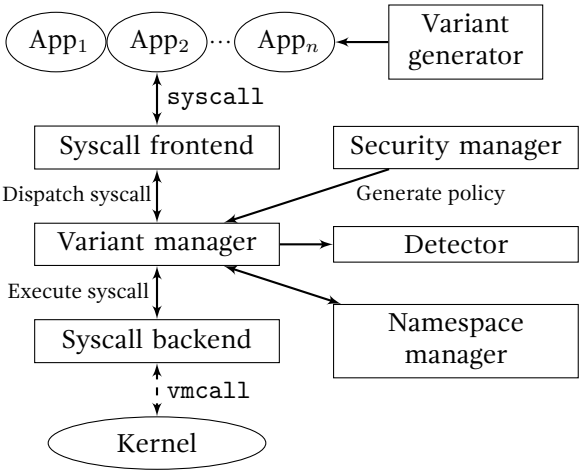


Figure 4.2: Overview of all MVX design components.

for other syscall interception methods by simply replacing the syscall frontend and backend components. At startup, the *variant generator* (Sec. 4.5.1) spawns an application instance for every variant, using an MVX-aware variant generation strategy in order to protect applications from all the previously mentioned attack vectors. The *security manager* (Sec. 4.5.2), in turn, generates security policies, providing a trade-off between security and performance for the rest of the components. These security policies can be defined by the user and depend on the classes of attacks considered.

When the application executes a syscall, it will trap into the *syscall frontend* (Sec. 4.5.3). This component is the entry point into both the monitor and the "kernel" (ring 0 code) of the virtual environment. The syscall frontend also manages all accesses to application state, such as its address space.

The frontend forwards all syscall events to the *variant manager* (Sec. 4.5.4), which is responsible for synchronizing all variants and enforcing the security policies. The variant managers communicate with each other using a ring buffer similar to that proposed by Hosek and Cadar [91]. Specifically, one of the variants (the leader) performs all of the actual syscalls and sends the corresponding events to the other variants (followers), who consume the events in their own time. The followers only execute a small set of these syscalls as well (e.g., memory management) as most I/O should happen only once (e.g., sending data over a socket). Unlike traditional MVX systems, the variants can run asynchronously most of the time, removing the great performance bottleneck of running variants in lockstep. However, unbridled asynchronicity is not safe. It would allow one variant to achieve arbitrary code execution with calls like `exec`, or information disclosure with calls

like `write`. Instead, we use the known distinction between security-sensitive and non-security-sensitive syscalls [165, 203] and enforce selective lock-step execution, where the set of sensitive calls varies depending on the security policy.

The variant manager is also responsible for deciding when a syscall should really be executed (leader) or when the results should simply be copied (followers). When the variant manager decides to execute a syscall, it sends it to the *syscall backend* (Sec. 4.5.5). In a naive implementation, the backend would simply forward all syscalls to the real kernel. However, each syscall in Dune requires a costly VM exit. To reduce these costs, we implemented a set of (memory management and `getpid`-like) syscalls directly in our monitor. Further libOS-style optimizations are also possible—for instance, by using a userspace network stack such as IX [19], or by batching syscalls [191].

The variant manager uses the *namespace manager* (Sec. 4.5.6) to ensure all information available to variants is the same (including PIDs, file descriptors, and timing information), and finally the *detector* (Sec. 4.5.7) to semantically compare the execution of the variants for divergence.

4.5 MvArmor: fast and secure MVX

We now describe each of MvArmor's components in detail.

4.5.1 Variant generator

A fundamental question in MVX is to what extent the variants should differ. Unconstrained variation makes it impossible to detect attacks from divergence, as *everything* may be different. Conversely, insufficient variation is also undesirable, as there may not be *any* divergence for an attack. Fortunately, for memory errors the straightforward solution is to vary the address space layout and keep everything else the same, as these differences should normally not affect program execution but will make a difference in the case of malicious memory actions. In this section, we identify several techniques that offer strong protection and detection against different classes of memory error exploits and we detail the corresponding implementation strategy in our current MvArmor prototype. For our analysis, we assume the now common PIE binary organization [199], but our design can, in principle, also handle non-PIE binaries by marking static program segments as non-relocatable and gracefully reduce security guarantees.

First of all, by using *non-overlapping address spaces* across variants (pioneered by [52]), any *absolute spatial* attack (i.e., attack relying on absolute code/data addresses) is already rendered ineffective. By ensuring that memory pages do not overlap across variants, a pointer can only be valid in at most one variant at a time and will thus *deterministically* crash all others. This already stops common

code-reuse attacks such as ROP [188] and information disclosure attacks such as JIT-ROP [190], as they rely on the absolute position of memory pages. In fact, even any other attempts (e.g., buffer overreads) to disclose code or data pointers will also cause divergence, as different values will be leaked to the attacker at the syscall level (e.g., over a socket) by construction. To enforce non-overlapping address spaces across variants, we randomize each variant using ASLR and then constrain ASLR not to reuse address ranges across variants. Since our MVX system resides in ring 0 of the virtualized environment, it has full control over the page tables, making ASLR modifications simple. MvArmor implements this technique for all the memory regions (i.e., code, data, stack, etc.) in each variant.

To also *deterministically* stop *relative spatial* attacks (i.e., attacks relying on relative code/data addresses), our variant generation strategy must be able to provide strong guarantees against buffer overflows/underflows and partial pointer overwrites. We observe that, for this purpose, our strategy must simply ensure that *offsets* between memory objects are non-overlapping. For example, if the size between objects on, say, the heap in a follower is as large as the entire (normal and compact) heap in the leader, any offset added to a pointer can only be valid in one of these variants. In other words, this design ensures *non-overlapping offset spaces* across variants, rendering all the relative spatial attacks ineffective. MvArmor implements this novel technique for all the heap objects, by using the standard "*compact*" allocator in the leader and a custom "*sparse*" allocator in the followers. Extending such guarantees to all the other memory objects is, in principle, possible, but source-level information is generally necessary to accurately decouple stack [114] and data [78] objects—although binary-level approximations are at times possible [43].

While the strategies described thus far can provide deterministic protection against all the spatial attacks, they are alone insufficient to stop *temporal attacks* (e.g., use-after-free exploits). Unfortunately, ensuring deterministic protection guarantees against generic temporal attacks is not practical without source-level information [62]. A practical binary-level alternative is to ensure *probabilistic* temporal safety guarantees. In our design, this is done by using different (randomized) memory allocators across variants and, to further limit the attack surface, by approximating type-safe memory reuse [5] at the binary level. MvArmor enforces probabilistic temporal safety for all the heap objects, randomizing the standard allocator with random inter-object gaps in the leader. In addition, MvArmor overapproximates type-safe memory reuse using per-size memory pools in our custom allocator in the followers (but much less conservative binary-level approximations based on allocation-time backtraces are also possible [5]).

While the implementation of all the proposed protection techniques can introduce significant overhead when all combined together on a single variant, their overhead can, in most cases, be completely masked across variants with our MVX

design. In MvArmor, followers are faster than the leader, as they do not execute most syscalls and thus waste several cycles waiting for the leader. Our measurements show that, for our baseline MvArmor implementation (i.e., without any protection enabled) on (I/O bound) server applications, the followers spend around 4,000 cycles on average per syscall waiting for the leader. Especially when syscalls do not require lockstep behavior, the idle periods leave sufficient time for the followers to spend more time in more expensive allocator abstractions implementing our protection techniques. This strategy provides strong security guarantees while reducing the run-time overhead of the end-to-end solution.

4.5.2 Security manager

The security manager generates policies that allow users to make trade-offs between security and performance. Specifically, a policy specifies whether each syscall is considered *non-sensitive* (event-streaming, meaning the leader can execute it without synchronization), or *sensitive* (requiring lockstep execution with other variants).

Security policies specify behavior at the level of the whole system, individual syscalls, or even specific arguments (e.g., "If the execute bit in a permissions flag is set then…"). In MvArmor, we propose the following policies for each of the aforementioned classes of attacks (but others are possible):

- *Code execution*: enforce full checks on `execve` and `mprotect`/`mmap` with execute permissions set.
- *Information disclosure*: enforce full checks on I/O syscalls that are able to leak data (e.g., `write`).
- *Comprehensive*: full checks on all syscalls.

In practice, the *Code execution* policy performs as efficiently as a policy where no syscalls are considered sensitive, since the syscalls considered by the *Code execution* policy are rarely executed in most applications.

The *Comprehensive* security policy, in turn, is useful to provide a generic catch-all strategy (and a lower bound on performance) but, given a target threat model, may provide comparable security to more tailored policies such as *Code execution* and *Information disclosure*. The key insight is that such security policies may delay detection of *failed* attacks, but they do deterministically and immediately stop successful attempts for all the attacks considered in the threat model.

4.5.3 Syscall frontend

When the application executes a `syscall` instruction, the execution will trap from ring 3 into ring 0—kernel space. By running the application and the monitor in a virtualized environment, all syscalls will trap into the monitor instead of the actual

kernel. MvArmor is based on Dune [18], which leverages virtualization to provide applications with access to privileged CPU features in a safe way. For this purpose, Dune relies on Intel VT-x extensions to allow a core to temporarily switch from the normal kernel (*VMX root*) into virtualized mode (*VMX non-root*) via the Dune hypervisor. Dune sets up ring 0 code for both *VMX root* and *non-root* mode, as shown in Figure 4.3. Since our monitor runs in privileged mode, it can also access other features, such as page tables and interrupts. While the same effect could be achieved by a kernel module or by modifying the kernel directly, MvArmor completely separates the monitor from the rest of the system, with no systemwide TCB increase.

The syscall frontend receives `syscall` traps from *libDune* and forwards them to the variant manager. In addition, the frontend is responsible for access to the application state, such as reads and writes to its address space.

Not all syscalls incur a trap on modern Linux systems; in every application, the kernel sets up a shared library (the virtual dynamic shared object—i.e., vDSO), which contains code to execute a selection of syscalls without trapping into the kernel. Using Dune, we can still intercept vDSO calls, by mapping our own code containing `syscall` instructions in place of the original vDSO.

4.5.4 Variant manager

Upon receiving a syscall event from the frontend, the variant manager synchronizes with the other variants. Every process has a ring buffer it shares with the respective processes in other variants. Upon completion of a syscall, the leader pushes it into the ring buffer together with its arguments and return value. The followers compare their arguments to those of the leader, and either execute the syscall themselves, or use the return value provided by the leader. Specifically, the variant manager has a per-syscall table to determine the appropriate behavior. Cer-

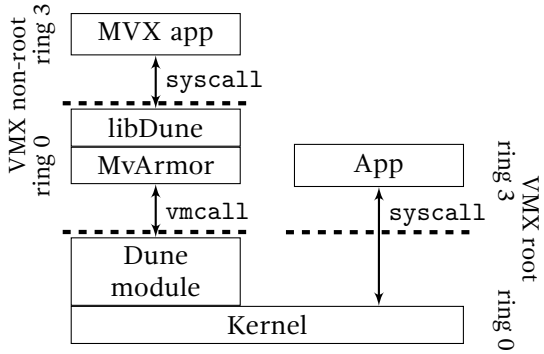


Figure 4.3: Control flow of syscalls with and without Dune.

tain syscalls should execute only once (e.g., socket-related syscalls), while others should execute in every variant (e.g., memory management calls). For the former, the followers simply copy the return value of the leader.

The ring buffers resemble those in Varan [91] and provide efficient communication without locking. We use atomic operations to update the ring buffer entries and busy-waiting to consume them. As syscalls like `select` and `epoll_wait` may block for a long time, followers stop busy-waiting after some time and go to sleep instead. Doing so is expensive due to the VM exits required for both sleep and wake-up calls. Assuming there is no shortage of cores, a more efficient solution is to sleep using Intel's `monitor`/`mwait` instructions as they incur no VM exit. Because our Dune-based virtualized environment runs in privileged mode, it can trivially use them where normal applications cannot.

As mentioned earlier, the security policy determines whether each syscall should run in lockstep, in which case the leader waits for all followers after pushing the arguments into the ring buffer. The followers then compare the syscalls as usual and then pause while the leader finishes the syscall. Doing so for every syscall (most conservative security policy) has a higher performance impact.

For multi-threaded applications, we force the followers to adhere to the order of syscalls of the leader. This strategy seeks to prevent divergent behavior due to non-deterministic scheduling decisions. This loose form of deterministic multithreading (DMT) was shown to be generally sufficient for previous MVX systems [91, 133]. Full DMT semantics could be used if issues do occur (such as divergence because of benign data races), at the cost of larger overhead [11, 60, 159].

4.5.5 Syscall backend

When a variant needs to execute a syscall, it forwards the call to the syscall backend. Forwarding syscalls to the kernel requires costly VM exits, so the syscall backend tries to execute the syscall locally when possible. This is currently implemented for memory management syscalls, but could be, for example, extended to include userspace network stacks such as IX [19]—which is also based on Dune. Besides eliminating VM exits, userspace network stacks also improve the overall performance of the application.

MVX monitors can be susceptible to time-of-check-to-time-of-use (TOCTOU) attacks, where an attacker modifies the arguments of a syscall in memory from a different thread after the arguments are checked by the monitor but before they are read by the kernel. This works because the arguments passed to the syscalls are usually pointers to buffers or structures in the address space of the application, copied separately by the monitor (for checking) and the kernel (for execution). We solve this by directly passing the pointers to the copied data structures (in the

monitor) to the kernel. Because no additional copying is required, this introduces no performance overhead.

4.5.6 Namespace manager

The namespace manager ensures the variants do not diverge accidentally by eliminating all variant-specific information. For instance, if variants were to have access to their kernel-assigned PIDs or timing information, they may (directly or indirectly) use such data in a conditional, leading to divergent behavior. The namespace manager therefore assigns virtual PIDs and TIDs to every process and thread using a hierarchical structure: when a variant creates threads in quick succession, these must get the same virtual TID in all variants, regardless of the order they actually appear on the system or the thread that executed its `clone` operation earlier.

We similarly virtualize file descriptors, as only the leader has access to all of them. For instance, followers do not have access to sockets or files opened as writable. Since the kernel assigns file descriptors in an incremental fashion per process and the followers open fewer files (e.g., read-only files) than the leader, these numbers start to diverge. The namespace manager therefore maintains a mapping of virtualized file descriptors to real (per-variant) file descriptors. The same holds for epoll-related identifiers, including the user data field. The `epoll_wait` syscall returns user-defined data previously registered when a socket has an I/O event. Since these values can be pointers (that differ per variant), they have to be mapped back to the socket and then to the variant-specific user data that should be returned for that socket.

Timing information should also not differ among variants, as this is often used for logging or seeding the random number generator. Since MvArmor has full control over the page tables of the application, it can easily intercept all vDSO syscalls. While not commonly used, we also disable the `rdtsc` instruction so that it traps into the monitor.

We ensure determinism in random number generation by allowing only the leader to open files like `/dev/random`. Pseudorandom number generators are generally seeded with information already virtualized by the namespace manager and require no additional effort to work correctly. We similarly limit access to the `/proc` file system to the leader. Without binary instrumentation, there is no easy way of interposing `rdrand` instructions. By disabling the corresponding bit in the `cpuid` implementation of the hypervisor, most normal applications and libraries will not use it (e.g., OpenSSL). While we have not observed the need to check on this further, we could also configure the virtual environment to trap into the hypervisor when the instruction is executed (via a bit in the control structure of the virtual environment).

4.5.7 Detector

Since a syscall can take no more than six arguments, many calls expect pointers to data structures which hold more information (e.g., a buffer or `struct`). For a full comparison between variants, the monitor therefore performs a deep semantic copy and comparison of such arguments [52, 133]. In MvArmor, the detector component performs both of these functions.

4.5.8 Implementation

MvArmor consists of a library implementing all the components in our design except for the frontend and backend. We developed two implementations of these components: our high-performance hardware-virtualization approach using the Dune sandbox and a `ptrace` implementation for development and debugging. These implementations call our shared library for every syscall and expose several functions such as how to access the monitored applications' address space and how to allocate memory across variants. The library itself consists of around 5,000 lines of C code, whereas the frontends and backends include around 500 lines of C code each.

The Dune sandbox, which is used to implement the default frontend and backend of MvArmor, allows for arbitrary applications to run in Dune. The sandbox loads a given binary using its own loader. It also implements bounds checking on any pointer passed to a syscall to prevent sandboxed applications from accessing ring 0 state such as the sandbox itself, the Dune library, or the monitor. We slightly modified both Dune and the sandbox to meet our requirements for the monitor, such as security fixes and more callbacks.

To implement the protection techniques discussed in Sec. 4.5.1, we used a modified version of `libumem`[2], a Linux userspace port of the Solaris slab allocator [27, 28]. This implementation served as a basis for our custom "*sparse*" allocator. In particular, by limiting the number of objects allowed (i.e., 1 object) per slab and adding padding (i.e., the leader's maximum heap size) to every slab, we enforce *non-overlapping offset spaces* between leader and followers. By preserving the natural per-size pooling architecture of `libumem`, we overapproximate type-safe memory reuse. Furthermore, since we want to retain the standard (randomized) allocator in the leader (to preserve security, but also performance guarantees in the slower leader), we assume both the standard and our custom allocator as trusted to prevent the monitor from detecting divergence caused by the different allocators (e.g., different syscalls to map memory). Note that while our custom allocator reduces ASLR entropy, this is irrelevant as our MVX security guarantees are stronger, and not ASLR-dependent.

[2] https://github.com/gburd/libumem

4.6 Limitations

At the time of writing, our MvArmor prototype has the following limitations:

- MvArmor's custom allocator is subject to Dune's restrictions on the maximum per-process virtual memory size, which currently requires relaxing the size restrictions on inter-slab padding (and thus security) in memory-intensive applications.
- While MvArmor can protect generic heap objects, it cannot decouple intra-struct buffers or chunks managed by custom memory allocators within each object without source-level information, a limitation fundamental to all the binary-level heap hardening solutions [5].
- While MvArmor's MVX library supports threading similar to recent MVX solutions [91], it cannot currently run multi-threaded applications. Extending our current MvArmor prototype to support arbitrary multi-threaded applications faces two challenges: (i) supporting thread safety in Dune (currently thread-unsafe), and, when benign data races are present (i.e., threads synchronizing without syscalls such as `futex`), (ii) preserving correct MVX semantics with a more strict form of DMT.

4.7 Evaluation

We evaluated MvArmor on a workstation with an Intel i7-3770 quadcore CPU clocked at 3.4 GHz and 16 GB of RAM. We disabled hyperthreading to eliminate (large) fluctuations in our test results. We ran all our experiments on a Debian 8.0 system, running a Linux kernel 3.2 (x86_64).

For our evaluation, we considered a number of popular server programs, which are heavily exposed to remote attacks (and thus would greatly benefit from the security guarantees provided by MvArmor) and have also been extensively benchmarked in prior work. In particular, we selected nginx (v0.8.54), lighttpd (v1.4.28), bind (v9.9.3), and beanstalkd (v1.10) for our experiments. We benchmarked bind, a popular name server, using the queryperf benchmark issuing 500,000 requests with 20 (default) threads. We benchmarked nginx and lighttpd, both high-performance web servers, using the wrk benchmark issuing 10 seconds worth of requests for a 4 KB page over 10 concurrent connections. We benchmarked beanstalkd, a work queue, with the beanstalkd benchmark issuing 100,000 push operations per worker over 10 concurrent connections and 256 bytes of data. To directly compare against Varan [91] (by far the fastest, but not security-oriented, state-of-the-art MVX solution), we adopted the same benchmark configurations (wrk and beanstalkd) considered in [91]—only increasing the number of push operations in the beanstalkd benchmark by a factor of 10 to

ensure a sufficient benchmark duration (i.e., 10-20 seconds).

We also evaluated MvArmor on microbenchmarks and on the SPEC CPU2006 benchmark suite, focusing our experiments on the CINT2006 benchmarks to reflect the configuration considered in [91] and provide comparative results. We ran all our experiments 11 times and report the median (with small standard deviation across runs). We report results for our default MvArmor configuration using a 10-element ring buffer (allowing the leader to execute 10 syscalls ahead of followers), but we observed similar results when moderately increasing/decreasing the ring buffer size. Unless otherwise noted, our experiments use the variant generation strategy from Sec. 4.5: the leader uses the default (randomized) allocator, whereas each follower uses the modified `libumem` allocator.

4.7.1 Server performance

To evaluate MvArmor's performance on our server programs, we first attempted to reproduce the over-the-network configuration described in [91], placing the client on a dedicated machine on the same rack as the server machine, with the 2 machines connected by a 1 Gbit/s ethernet link. In our setup, this configuration was insufficient to effectively saturate the server, reporting only marginal performance impact across all our programs. To fully saturate all our server programs, we then reran our benchmarks through the loopback interface. We note that this strategy resulted in sound but also more pessimistic performance results. As an example, the original Dune paper reports ∼1% overhead on lighttpd for an over-the-network configuration [18]. Reproducing the exact same experiment in our setup (which, in contrast, relies on the loopback interface) resulted in much higher impact (∼12%). This is caused by less networking overhead, making the impact of our system more visible.

We evaluated MvArmor's performance across all the security policies supported. Figure 4.4 shows the results for the *Code execution* security policy for an increasing number of variants (1 through 4, where 2 variants means 1 leader and 1 follower). Since the server applications do not normally execute any syscalls that fall under the *Code execution* policy, the results are identical to a policy where no syscalls are considered sensitive. When disabling our variant generation strategy (isolating the MVX synchronization overhead) for both policies, we observed no differences in the results. This shows that the followers are indeed able to keep up with the leader despite the less efficient secure allocator, hence our variant generation strategy has essentially no impact. Note that security policies have no effect on scenarios with only one variant, as there is no synchronization in such scenarios.

Figure 4.6 reports results for a slightly more conservative policy (*Information disclosure*). As our server applications make heavy use of write syscalls, which

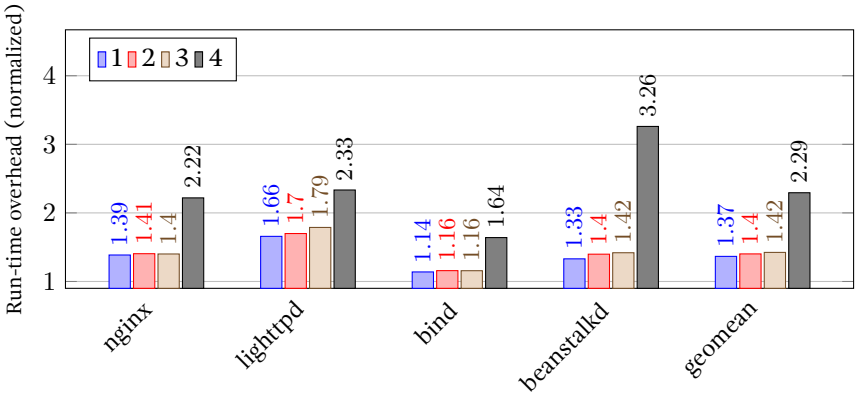


Figure 4.4: Overhead using the *Code execution* security policy for increasing number of variants.

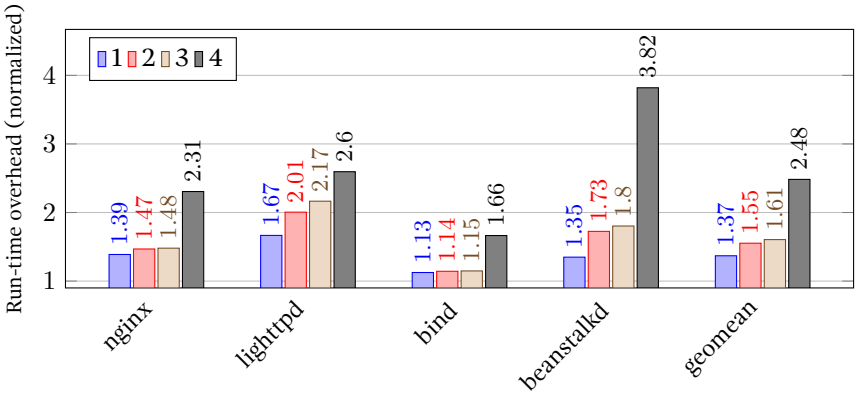


Figure 4.5: Overhead using the *Comprehensive* security policy for increasing number of variants.

run in lockstep now (forcing idle time in the followers to be spent waiting for the leader rather then performing extra operations), the overhead of the slower followers (and of our variant generation strategy) cannot be completely masked in this case. For comparison purposes, Figure 4.7 reports results for the same policy, but with our variant generation strategy disabled. Finally, Figure 4.5 reports results for our most conservative security policy possible (*Comprehensive*, generally overly conservative, but useful to provide worst-case results). In this configuration, the impact of our variant generation strategy (and custom allocator in the followers) is more noticeable given that all the syscalls run in lockstep (e.g., 55.2% vs. 49.1% for two variants, geomean).

As shown in the figures, programs with a lower number of "copy-heavy"

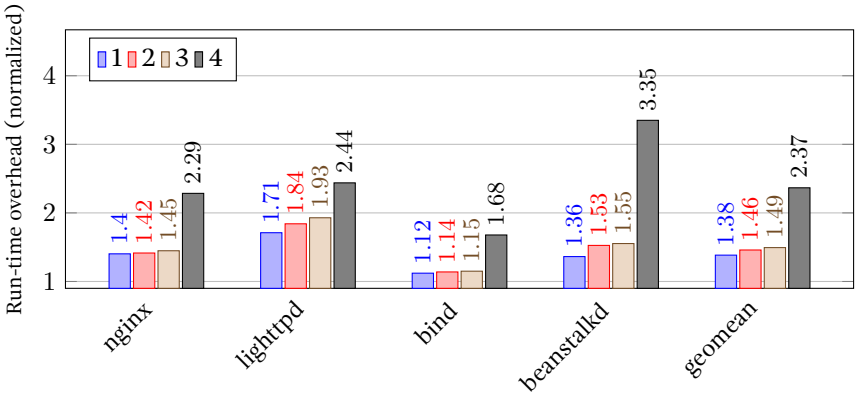


Figure 4.6: Overhead using the *Information disclosure* security policy for increasing number of variants.

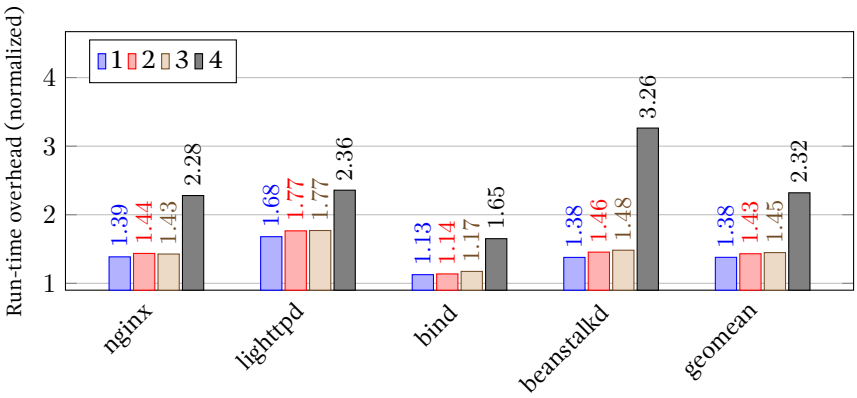


Figure 4.7: Overhead using the *Information disclosure* security policy for increasing number of variants with our variant generation strategy **disabled**.

syscalls such as nginx and lighttpd scale less efficiently with the number of variants and are also more affected by the increasingly lockstep-like behavior enforced by more conservative security policies (e.g., full lockstep for *Comprehensive*). Beanstalkd, on the other hand, issues relatively few syscalls overall and the reported overheads are mostly due to the copying costs for Beanstalkd's large buffers. This results in Beanstalkd performance being non-trivially impacted by our syscall interposition strategy, but scaling well with the number of variants and with more conservative security policies. We observed similar behavior for bind, which only issues 2 very "copy-heavy" syscalls per request (i.e., `recvmsg` and `sendmsg`).

Overall, our results suggest that MvArmor scales well with the number of variants, for example with only a ∼3% average overhead increase (geomean) when moving from a 2-variant to a 3-variant configuration (across different security policies). We observe significant performance drops only when exhausting the number of cores—a 4-variant configuration in our 4-core setup, with a core dedicated to the benchmark program. We also note that a more large-scale scalability analysis, while possibly interesting, is irrelevant in practice. MvArmor's default configuration using 2 variants is sufficient to provide strong security guarantees by construction and also minimizes core utilization to encourage deployment in real-world settings.

We now compare our results against Varan [91]. For a fair comparison, we focus on the *Code execution* security policy, which most closely matches Varan's MVX strategy. MvArmor's performance is very similar to Varan for nginx (41% vs. 37% for 2 variants, respectively), but we did observe somewhat different results for beanstalkd and lighttpd. In particular, for beanstalkd, our results show relatively low and stable overheads for MvArmor (∼41%) and increasingly higher overheads for Varan (52% and 57%, for 2 and 3 variants, respectively). Conversely, for lighttpd, Varan reports low and nearly constant overheads (∼14%), while MvArmor's overheads start at 70% for 2 variants.

Overall, MvArmor's overhead results are generally comparable to Varan, although our experiments show that the actual performance of the 2 systems depends on the program considered. We believe our results are very encouraging, given that (i) Varan is the fastest existing MVX implementation, (ii) compared to Varan, MvArmor provides much stronger security guarantees even for our least conservative (*Code execution*) security policy, (iii) our loopback-based performance results are pessimistic and could also be improved by operating further libOS-style optimizations enabled by our design.

4.7.2 SPEC performance

To further compare our results against prior solutions, we evaluated the performance impact induced by MvArmor on the SPEC CINT2006 benchmarks. While the SPEC benchmarks are CPU-intensive and issue a relatively low number of syscalls (thereby providing optimistic performance results for MVX systems), this experiment still provides useful comparative data points. Figure 4.8 presents our findings.

As shown in the figure, MvArmor yields an average overhead of 9.1% (geomean) for 2 variants and 20.4% for 4 variants across all the benchmarks. A closer inspection revealed that, in most cases, the reported overheads primarily originated by the impact on the memory bandwidth of the system. Benchmarks such as `mcf` and `libquantum` are particularly memory-intensive and tend not to scale well in simul-

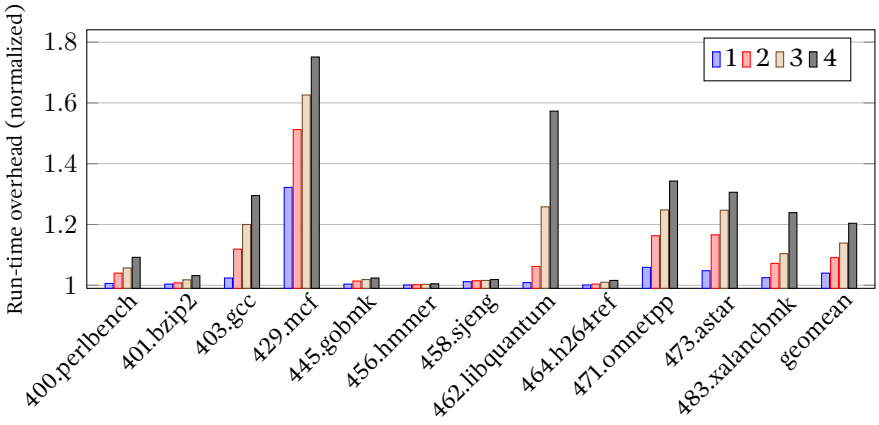


Figure 4.8: Overhead for the SPEC CINT2006 benchmarks for an increasing number of variants.

taneous runs on multi-core architectures [98].

Nevertheless, despite the greater impact of TLB misses in memory-intensive benchmarks induced by the use of EPTs in Dune [18], our results are encouraging and, in fact, even yield better performance than Varan, the best MVX performer on SPEC in the literature, reporting an average overhead of 14.2% (geomean) with 2 variants [91] on the same set of benchmarks.

4.7.3 Microbenchmark performance

To carefully pinpoint the sources of overhead introduced by MvArmor, we evaluated our solution using a number of microbenchmarks. In particular, we measured the number of cycles required by various syscalls from the perspective of a user program while running under Dune [18] and under our full MvArmor solution with 1 or 2 variants (MV1 and MV2, respectively). Figure 4.9 presents our findings.

Both the `getpid` and `close(-1)` syscalls have a very short duration, with the kernel almost immediately returning to userland. For these simple syscalls, Dune alone adds around 1,300 cycles, accounting for syscall interposition and (mostly) for the `vmcall` to the host. MvArmor, in turn, adds around 300 extra cycles on top of Dune. Our microbenchmark results are compatible with those in the original Dune paper [18]. As the `getpid` syscall is currently implemented in MvArmor's backend, it does not require an expensive `vmcall`. This is reflected in the significantly reduced overhead compared to simply running Dune in passthrough mode (783 vs. 1,513 cycles), even with the additional MVX logic in place. This shows libOS-style optimizations are a viable strategy to speedup MvArmor in the future.

The `write` syscall has a buffer argument, which first undergoes bounds checking in Dune and then requires copying the buffer in MvArmor's monitor.

For a small 5-byte buffer, Dune alone adds around 1,500 cycles, while MvArmor adds around 400 (`/dev/null`) and 800 (filesystem) extra cycles. When writing to `/dev/null`, the kernel does not have to wait for I/O, as opposed to the case of writes to the filesystem. Since the overheads added by Dune and MvArmor are fairly constant, the overall performance impact quickly becomes insignificant with more lengthy I/O requests—such as those typically issued by server programs.

The `time` syscall, part of the vDSO, can natively be executed without a `syscall` instruction, but Dune remaps the vDSO to force traps into the monitor. While this strategy introduces a non-trivial performance impact (around 1,050 cycles for Dune alone), it also allows MvArmor to monitor and alter its return value to ensure consistent variant behavior.

4.7.4 Security

To analyze the effectiveness of our variant generation strategy against memory errors, we present an analytical security analysis on different classes of exploits and draw from real-world examples. We note that, since an empirical evaluation of security against arbitrary existing exploits would have trivially detected deviations (and thus attacks) in all cases, we opted for an analytical analysis similar to prior work on randomization-based solutions [21, 78].

First, memory error exploits that rely on absolute addresses are *deterministically* prevented by MvArmor's non-overlapping address spaces across variants, regardless of the particular security policy deployed. These exploits can be used to mount many classes of attacks, ranging from modern code-reuse attacks [183, 188] to information disclosure attacks [185].

To exemplify the security guarantees provided by MvArmor for these classes of attacks, we consider an exploit based on a real-world vulnerability (CVE-2004-0488). This vulnerability allows an attacker to mount a stack-based buffer overflow exploit against Apache httpd, corrupting a data pointer with an absolute address and granting the attacker the ability to read arbitrary memory values [185]. While this attack is effective against 1 standalone variant (assuming the attacker can bypass ASLR [190] and disclose the intended absolute memory address), an attacker will not be able to find a single absolute memory address which is, at the same time, valid across 2 variants running in parallel—resulting in at least 1 protection fault and MvArmor detecting the attack.

Memory error exploits that rely on relative addresses are also *deterministically* prevented by MvArmor's MVX-aware allocator design deployed in the follower(s). These exploits can be used to mount many classes of attacks, e.g., information disclosure/tampering, and other non-control data attacks [92].

To exemplify the security guarantees provided by MvArmor for these classes of attacks, we consider two real-world exploits crafting relative memory read and

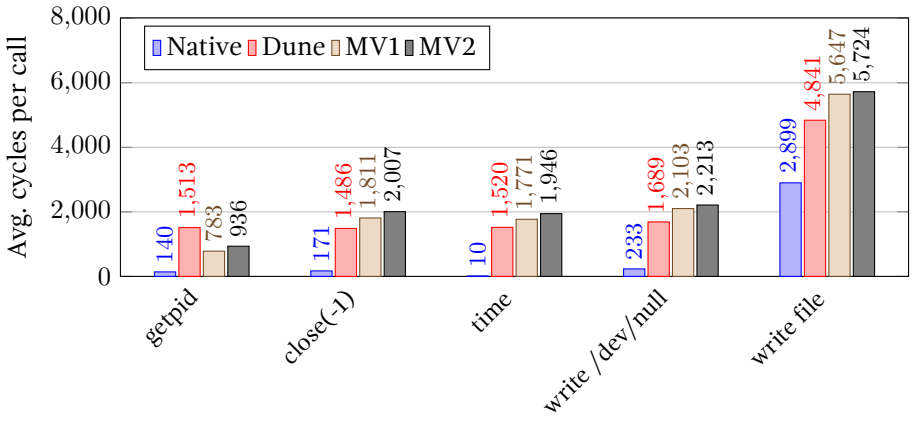


Figure 4.9: Average number cycles for various syscalls. Dune (using the sandbox app) always runs in passthrough mode. MvArmor runs with 1 (MV1) or 2 (MV2) parallel variants using the *Code execution* security policy.

write primitives to achieve the attacker's goals (respectively). We also speculate on an attacker extending these exploits by using temporal vulnerabilities, to demonstrate how MvArmor would *probabilistically* prevent more advanced attacks.

For the former case, we consider an exploit based on the Heartbleed vulnerability in the OpenSSL library (CVE-2014-0160). The exploit overreads a buffer located on the heap to read security-sensitive data from other heap objects and eventually allow the program to leak them over the network. With MvArmor deployed, the attacker can only read data from the leader because of the non-overlapping offset spaces. Any attempt to read the object in the follower(s) as well would require reading a size larger than the leader's heap, causing the leader to read past its heap and crash. Even if an attacker were to find, say, a use-after-free vulnerability to leak the same security-sensitive data, MvArmor would still probabilistically stop the attack. Since the data is leaked using standard I/O syscalls, MvArmor's *Information disclosure* security policy can immediately identify the divergent behavior of reading probabilistically different data from different objects in the leader and the followers (due to the different and randomized allocators, as well as type-safe reuse in the follower(s) increasing the gap), and thus still detect information disclosure.

For the latter case, we consider an exploit based on another vulnerability in the OpenSSL library (CVE-2014-0195). The vulnerability allows an attacker issuing a long non-initial fragment to overflow a heap-allocated buffer and corrupt adjacent data. The exploit relies on this primitive to corrupt security-sensitive non-control data in other heap objects. With MvArmor deployed, the attacker is, again, forced to overflow more data to compensate for the inter-object gaps on the heap in the

follower(s) and reach the intended security-sensitive data (e.g., UID) across all the variants. However, any attempt to "spray" this much data would again result in protection faults in at least one of the variants, as described earlier. Similarly, even if an attacker were to find, say, a use-after-free vulnerability to tamper with the same security-sensitive data, any write will, again, likely result in different side effects across variants and probabilistically stop the attack.

Finally, since MvArmor captures deviations in external behavior by monitoring differences in security-sensitive syscall patterns, the detection guarantees provided against such attacks improve with the conservativeness of the security policy deployed. Note that when not deploying our most conservative security policy (*Comprehensive*), MvArmor may fail to detect some *failed* attack attempts immediately, but will still detect (and disallow) all the behavioral deviations induced by successful attack attempts that affect security policy-defined syscalls.

4.8 Related work

The idea of using software diversity to improve fault tolerance was first introduced by Avižienis and Chen [12] in the seventies. Their idea of N-version programming had multiple teams of programmers implementing the same software, hoping bugs would be isolated to only one of the versions. This paradigm was only later expanded to security applications [101]. In 2006, Cox et al. [52] introduced the idea of using automatically generated variants, rather than versions, to improve deployability. They also analyzed multiple monitoring strategies, such as syscalls (using a kernel module) and a network proxy, and several variant generation strategies, such as disjoint memory mappings and instruction set randomization. DieHard [20], also published in 2006, proposed a probabilistic memory safety solution including a "replicated mode". While similar in spirit, MvArmor provides far better performance and security. Variant synchronization issues, in turn, have been the focus of extensive research ever since [33, 90, 133, 180, 209].

Salamat et al. [180] describe other variant generation strategies in Orchestra, proposing a reversed stack in one variant to prevent stack smashing attacks. Orchestra relies on the `ptrace` API to interpose syscalls, similar to most existing MVX monitors [33, 90, 133, 209]. In 2015, Hosek and Cadar proposed Varan, which relies on static binary instrumentation in order to significantly improve MVX performance. In contrast to MvArmor, Varan focuses on software reliability rather than security, similar to other MVX-like systems such as Tachyon [133] and Mx [91].

Varan's event-streaming design shares similarities with MvArmor's variant synchronization strategy, in that they are both based on a ring buffer design inspired by existing high-performance lock-free ring buffers [85, 197]. The key difference is that Varan's event-streaming design is fully asynchronous (other than not isolated from the untrusted program execution) and unable to

support the synchronous detection policies employed by MvArmor's design for security. Varan's event-streaming architecture shares, in fact, similarities with record-and-replay systems, in which the execution is continuously recorded into a log. This log can later be used to reexecute the application, locally or on a different machine, and optionally deploy additional checks during replay, for example for security auditing purposes. An example in this category is Paranoid Android, a record-and-replay system which can efficiently deploy even heavyweight security analyses when replaying mobile apps' execution in the cloud [176].

The idea of combining ASLR [169] with MVX, allowing for non-overlapping layouts to combat code-reuse (and other absolute address-based) attacks, was first proposed by Cox et al. [52]. MvArmor extends these techniques to build a new MVX-aware variant strategy which allows complementary per-variant allocators to control memory object allocation in a fine-grained way and effectively counter arbitrary memory error exploits that rely on both absolute and relative object locations. MvArmor could be complemented with other memory layout modification strategies, such as fine-grained randomization. Fine-grained randomization [21, 78] has been previously proposed as a comprehensive defense solution against arbitrary memory error exploits, but has proven to be ineffective on its own against modern information disclosure attacks that can bypass any form of ASLR altogether [190].

Finally, the use of hardware-assisted virtualization to sandbox individual processes (granting them access to privileged CPU features) was first proposed by Dune [18], which also forms the basis of MvArmor.Hardware-assisted virtualization has also been used to isolate parts of the operating system itself [152] and to facilitate libOS implementations [16, 19, 172]. MvArmor draws from prior research in both directions, on one hand, relying on virtualization to efficiently and securely isolate the MVX monitor from untrusted execution, and on the other hand, exploiting libOS-style optimizations to further mitigate the performance impact of traditional MVX implementations.

4.9 Conclusion

In this chapter, we presented a new design for secure yet efficient MVX systems. Our MVX monitor design leverages hardware-assisted process virtualization to securely and efficiently gain full control over the running program variants. We complemented our design with a new MVX-aware variant generation strategy, which improves the performance and security guarantees of all the prior MVX proposals, resulting in a much more efficient and comprehensive defense solution. Our end-to-end design effectively combines the comprehensive protection against arbitrary memory error exploits provided by fine-grained ASLR strategies with the strong attack detection and disclosure-resistant guarantees provided by MVX.

We implemented our ideas in MvArmor, a new secure and efficient MVX system. MvArmor demonstrates that many of the performance and/or security limitations of existing MVX solutions are not fundamental and can be effectively addressed with a careful design. MvArmor's policy-driven detection strategy can provide strong and flexible security guarantees at the cost of relatively low runtime overhead for such a comprehensive security solution. Even more surprisingly, MvArmor can match the performance of the fastest MVX implementation available while providing far stronger security. Finally, based on a design particularly amenable to optimizations, we believe our framework can provide new opportunities to further enhance the performance of MVX systems. To foster further research in the area and in support of open science, we are making our MvArmor prototype available as open source, available at `http://github.com/vusec/mvarmor`.

5 kMVX: Detecting Kernel Information Leaks with Multi-variant Execution

Kernel information leak vulnerabilities are a major security threat to production systems. Attackers can exploit them to leak confidential information such as cryptographic keys or kernel pointers. Despite efforts by kernel developers and researchers, existing defenses for kernels such as Linux are limited in scope or incur a prohibitive performance overhead.

In this chapter, we present kMVX, a comprehensive defense against information leak vulnerabilities in the kernel by running multiple diversified kernel variants simultaneously on the same machine. By constructing these variants in a careful manner, we can ensure they only show divergences when an attacker tries to exploit bugs present in the kernel. By detecting these divergences we can prevent kernel information leaks. Our kMVX design is inspired by multi-variant execution (MVX). Traditional MVX designs cannot be applied to kernels because of their assumptions on the run-time environment. kMVX, on the other hand, can be applied even to commodity kernels. We show our Linux-based prototype provides powerful protection against information leaks at acceptable performance overhead (20–50% in the worst case for popular server applications).

5.1 Introduction

With millions of lines of code, the operating system (OS) is typically one of the most complex pieces of software on a machine. All the research into alternative OS designs and safer languages notwithstanding, monolithic kernels such as Linux, Windows, and BSD (all written in unsafe languages), are still the norm. Unfortunately, such kernels offer a large attack surface. For instance, the Linux kernel contains a long list of exploitable bugs [39, 104, 130, 161, 171, 217], with the most common class of vulnerabilities being that of *information leaks* [130]. Such vulnerabilities allow an unprivileged attacker to extract sensitive information such as crypto keys or pointers from the kernel. Not only do these bugs compromise the confidentiality of the kernel, but they are often critical to subsequent attacks such as privilege escalation [195].

While developers and researchers have proposed numerous defenses and detection tools in response [104, 108, 121, 123, 128, 163, 167, 175, 182], their solutions tend to be either too limited in scope and detection capability, or too expensive in terms of overhead. For instance, kernel address space layout randomization (kASLR) in its current form only randomizes the base of the kernel, meaning a single leaked pointer compromises the entire randomization. The built-in KASAN memory error detector [123] can only detect use-after-free and out-of-bounds bugs, at the price of a 4x performance overhead. Meanwhile, kmemcheck [121], another built-in memory checker, can also detect uninitialized reads, but at the cost of several orders of magnitude of overhead. More recent efforts, such as UniSan [128], efficiently mitigate uninitialized reads, but does not cover all the other information leaks.

In this chapter, we present a new technique, called kMVX, which can efficiently detect arbitrary information leaks by running multiple diversified kernels simultaneously on the same machine. These *variants* are constructed in such a way that they exhibit the same behavior in normal circumstances, but show *diverging behavior* if an attacker tries to exploit the system. We achieve this by making particular changes in the memory layout of each variant, which make no difference during benign execution, but do matter for an attacker trying to leak information from the kernel. While the *variant generation* provides the security, our kMVX design also requires components to facilitate running multiple kernels on the same machine and detect the divergences.

The design of kMVX is based on that of multi-variant execution (MVX) [20, 52, 107, 180, 208]. MVX is a user-level defense that runs multiple variants of an application side-by-side and checks their behavior. Traditional MVX synchronization principles are not applicable to kernels, however, as they often lack such clear and strict interfaces, can interact with system resource and hardware directly, and contain many sources of non-determinism (e.g., task scheduling). kMVX addresses

this challenge by introducing two synchronization points, called the *I/O sync* and *syscall sync*.

We constructed a kMVX prototype based on Linux, which shows our kMVX design applies to commodity operating systems and is effective at detecting kernel info leaks. Our prototype runs multiple Linux kernels on the same machine while preserving the user space ABI of Linux, allowing existing applications to run unmodified on the system. Experiments show our prototype has at most 20–50% overhead for real-world server applications.

To summarize, our contributions are three-fold:

- A design for kMVX that supports running and monitoring multiple kernel variants on a single system.
- Multiple variant generation strategies applicable to the kernel to stop information leaks.
- An evaluation of a kMVX prototype on Linux that demonstrates its effectiveness and efficiency.

5.2 Background

OS kernel vulnerabilities Similar to most large projects written in unsafe languages, the Linux kernel contains a large number of bugs, with more being discovered—and even introduced—every month. Despite the numerous efforts to add detection and protection mechanisms, studies show that there is still a wide variety of bugs present in the kernel [39, 128]. For instance, in their 2010 study, Chen *et al.* [39] identified four major classes of exploits in the Linux kernel: memory corruption, policy violation, denial of service, and information leaks. The latter was, and still is, by far the most dominant class of vulnerabilities [128].

This motivates our focus on *information leak* attacks in which attackers abuse such vulnerabilities to induce kernel data to leave the kernel, for instance to be sent over a socket or copied to user memory. A common cause of such info leaks are data structures where some fields are uninitialized before passing it to the network stack or userland. A recent study shows that some of these info leaks are also present in data structures where the compiler inserts padding bytes that are not initialized, both on the stack and the heap [128].

As an example, both CVE-2014-1444 and CVE-2016-4569 concern a data structure allocated on the stack where the compiler adds padding, and can thus leak information to user space when called via an `ioctl` syscall (see Listing 5.1). If a previous stack frame contained sensitive data, such as kernel pointers, these will be copied alongside the data structure. CVE-2013-2237 is an example where an

```
struct snd_timer_tread {
  int event;                  // + 4 bytes padding
  struct timespec tstamp;
  unsigned int val;           // + 4 bytes padding
};
int snd_timer_user_params(...)
{
  struct snd_timer_tread tread;
  tread.event = SNDRV_TIMER_EVENT_EARLY;
  tread.tstamp.tv_sec = 0;
  tread.tstamp.tv_nsec = 0;
  tread.val = 0;
  // Padding uninitialized at this point
  // ...
  copy_to_user(usr_buffer, &tread,
          sizeof(struct snd_timer_tread));
}
```

Listing 5.1: CVE-2016-4569: The compiler adds several bytes of padding to the `struct snd_timer_tread` in `sound/core/timer.c`. The `copy_to_user` call will leak the uninitialized padding bytes to the user.

object is allocated on the kernel heap without being fully initialized, and then sent over a socket, again leading to an info leak.

While info leaks themselves may be already harmful, they are often also used for further attacks. For instance, when kASLR is enabled, an attacker typically first has to leak pointers before mounting more complex exploits. An example is CVE-2016-0728, a use-after-free bug in the keyring management. An attacker can force the kernel to de-allocate an object while it still holds pointers to it. The attacker can then allocate a new object (of a different type), which the kernel will interpret as the old object. In this particular case, an attacker can place arbitrary kernel pointers inside the object which the kernel will then call, leading to a privilege escalation. In cases where kASLR is enabled, the attacker first needs to determine the correct pointers for the last step of this attack via an info leak.

Multi-variant execution kMVX draws from user space MVX, which has been applied to programs ranging from servers to graphical applications [20, 52, 91, 107, 173, 174, 180, 209]. The core idea of MVX is to generate variants that have the same *outside* behavior (or output) given the same inputs in normal situations, but start to diverge when an attacker tries to exploit a vulnerability in the program. A simple example is running the same program twice with address space layout randomization (ASLR) enabled. ASLR variations have no observable effect on the program execution from the outside: when the same input is applied to both vari-

ants they will return the same output. However, if the application contains a bug that allows an attacker to trigger a read at a specific memory location, via a pointer, the application may return arbitrary memory content, hence we will observe divergent behavior. To be specific, both variants will receive the same pointer, but due to ASLR the pointer will most likely have different contents in each variant.

The security guarantees of MVX and kMVX systems are determined by the diversification strategies of the *variant generator*. The design of MVX and kMVX allows for variant generation strategies to be swapped and combined, based on the threat model. Schemes with stronger (deterministic) guarantees often require large amounts of resources [107] or introduce compatibility issues [180]. One important insight for MVX and kMVX is that the amount of entropy itself does not provide the security (like it does for ASLR), instead the security comes from the entropy causing *some* divergence in execution for attacks. Creating suitable variant generation strategies for kMVX, that work with the strict requirements of kernel resources and provide full coverage against information leaks, is a key challenge we address in the chapter.

When running multiple variants, it is crucial both have the same view of the outside world to avoid benign divergences. For MVX this is relatively easy, since user space programs have a strictly defined I/O interface in the form of syscalls. For instance, both variants should have the same view of time and network traffic, which are both accessed via syscalls. The kernel does not have such an interface, creating a number of challenges for kMVX, which we address in Section 5.4.

5.3 Threat model

MVX in general can be applied to a large number of different exploits targeting applications. All the related vulnerabilities can be found in the kernel as well. As kMVX is a new design, for this chapter we primarily focus on attacks on a *local* system, where *information is leaked* via bugs in the kernel code.

We assume a *local* attacker already in full control of an unprivileged user space program, who tries to disclose pointers or sensitive information (e.g., cryptographic keys) from the kernel by (repeatedly) interacting with it via syscalls and exploiting info leak vulnerabilities. We focus on exploitation of such vulnerabilities via software bugs and assume orthogonal defenses for other attack vectors such as side channels.

Due to defenses such as kASLR [68], a majority of Linux kernel attacks rely on leaking information at some stage in the attack. By eliminating info leaks, we hinder many other classes of exploits, such as privilege escalation.

5.4 kMVX: Kernel multi-variant execution

Our design of kMVX runs two co-existing OS kernels simultaneously on the same (virtual or physical) hardware. Each kernel is diversified using our *variant generator* (*vargen*) to cause divergence for information leaks. Each kernel includes two components to both keep execution consistent and check for divergences caused by malicious users: the *I/O sync* and *syscall sync*. These two components are similar in spirit to the monitor of traditional MVX. An overview of the components of kMVX can be seen in Figure 5.1.

The I/O sync part, placed at the hardware-kernel interface, is responsible for preventing spurious divergences by providing a uniform interface to the (shared) hardware. For instance it provides both kernels with the same view of time and the network. Information leaks are detected by divergent behavior at the user space boundary in the syscall sync. Given the local attacker in our threat model, the I/O sync does not need to check for divergences, simplifying the design.

kMVX follows the conventional leader-follower design for MVX [91, 107, 208]. In Figure 5.1, kernel 1 is the leader and performs actual I/O operations. The other kernel is the follower, which synchronizes with the leader via an in-memory communication channel.

For variant generation, we ensure the virtual memory address spaces of the two kernels do not overlap. In addition, we diversify the different allocators in the kernel and randomize the stack usage with a compiler pass. For instance, we modified the `kmalloc` allocator in Linux to be type-based, both to prevent use-after-free attacks but also to generate a different memory layout for objects between the variants. All these diversifications together provide us with a strong defense against info leaks through software bugs.

Although running multiple kernels is also possible via *virtualization*—running each kernel variant in a separate virtual machine—this is not sufficient for kMVX. The level of synchronization required for kMVX requires more in-depth knowledge of the kernel state and its ordering (e.g., scheduling between cores). Similarly the syscall sync, in particular data transfers between user space and the kernel, would require kernel modifications to work properly with a virtual machine monitor (VMM). Overall, the kernels still require significant modifications despite using a VMM, without virtualization giving any significant benefits or portability. As such, we do not use virtualization for our kMVX design.

In kMVX both kernels are partitioned in physical memory, and limited to certain CPU cores. Only the leader has access to all the hardware on the machine, such as the network interface, whereas the followers have to communicate with the leader to interact with the outside world. In our kMVX design, our aim is to place the I/O sync as close to the hardware boundary as possible, maximizing the kernel code executed in each kernel.

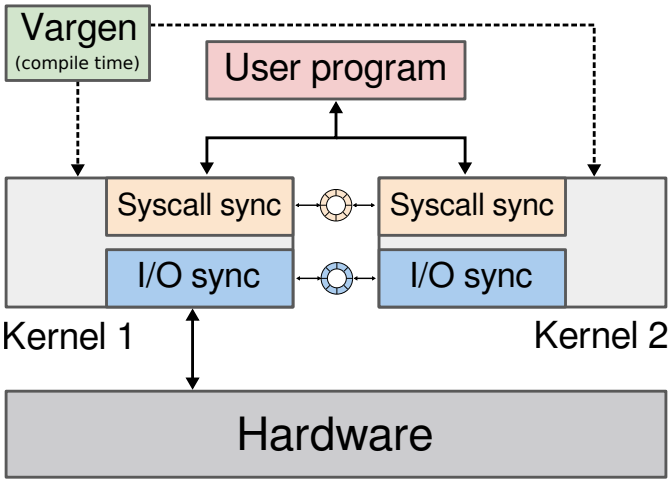


Figure 5.1: Overview of the kMVX design. Two kernels run on the same hardware by controlling interactions with the hardware via the *I/O sync* component. Interactions with user space go through the *syscall sync* to detect divergences. *Vargen* constructs diversified kernels during compilation.

5.4.1 Syscall synchronization

The syscall sync component at the user space boundary is responsible for detecting sensitive information leaking to the attacker. Because of the variant generation, such information will differ when returned to user space, which is then detected. Operating systems have a strict interface for copying data to the user (in Linux there are `copy_to_user`, `put_user`, and the syscall return value) which our syscall sync interposes. Any data copied to the user is placed in the shared ringbuffer to be validated for divergence. If any divergence is detected the faulty bytes are zeroed, to eliminate the info leak without stopping execution (but logging the event). This is especially important for compatibility, since bugs may cause non-malicious info leaks during regular execution.

Because data can only be returned after being checked and (potentially) being zeroed, this effectively enforces lockstep behavior between the variants for copying data to the user. Previous user space MVX designs had a notion of *policies*, where only certain (critical) syscalls are running in lockstep, allowing the user to tune the trade-off between security and performance [91, 107, 208]. Our kMVX design effectively enforces such a lockstep policy since syscalls that copy data to the user are considered *critical syscalls*.

Placing the monitor inside the variants themselves does not compromise the security of our design, since it is mapped at a different location in both kernels (Section 5.4.3). Any access to this area first requires the attacker to leak its location, which is prevented by the kMVX monitor itself.

5.4.2 I/O sync

In order to implement kMVX, we need multiple variants of the OS kernel running simultaneously on the same hardware. However, commodity operating systems such as Linux are clearly not intended to run multiple kernel instances. More importantly, when multiple kernel instances natively run on the same hardware, the management of the hardware state, specifically device state—a task performed only by the kernel—must be done carefully. Note that when applying MVX to user space programs, the I/O interface is clearly defined with syscalls. However, for drivers and their interactions with hardware there is generally no such interface [109].

If two kernels share the same set of cores, the execution may need to be serialized, requiring the leader to wait a significant time for the follower to finish before finishing synchronization. Instead we parallelize the implementation, associating a fixed partition of cores to each kernel. This allows both kernels to execute at the same time, greatly reducing the waiting time required for synchronization.

5.4.3 Variant generation

For our variant generation, we modify existing kernel allocators to yield different usage patterns between variants. In particular, we apply variant generation by partitioning the address space, modifying dynamic allocators such as the SLAB allocator, and changing the format of the stack with a compiler pass. Variant generation is, in some cases, highly dependent on the software. In this section we focus on Linux, but all techniques have more general applicability.

Address space partitioning Partitioning of the address space is a well-known variant generation scheme for MVX that ensures any pointer can only be valid in at most one variant at a time [52]. Since kernel pointers are a common target for info leaks, making sure these always differ between variants means they are impossible to leak, since that immediately causes a divergence.

Userspace programs are characterized by a very large address space: the OS provides programs with a virtual address space that can span up to 128 TB on x86-64. Our experiments show that less than 1% of that is virtually mapped in a process. Moreover, all regions mapped into userland (e.g., stack, heap, libraries, binary) are very small on their own and can be arbitrarily placed anywhere in the address space during runtime. The kernel address space layout, on the other hand, is strictly defined and determined at compile time.

Luckily, because virtual address ranges are much larger than the actual physical memory-addressable resources of most computers, the 128 TB of kernel virtual address space on x86-64 can be split into two separate partitions without significant drawbacks, enabling address space partitioning of the kernel memory. By doing so, we ensure that any leaked pointer will differ and trigger detection. For

partial pointer leaks, where only some of the bytes of a pointer are leaked, we add additional entropy on each variants relying on the existing kASLR mechanisms. Recall that the amount of entropy is not very important, since we just needs *some* variation.

Dynamic allocation OS kernels such as Linux generally offer several ways of dynamically allocating memory. For Linux, of particular interest are the SLAB allocator and `vmalloc`. With `vmalloc` it is possible to allocate one or more pages, which are virtually but not physically consecutive, and allocations are always rounded to a multiple of the page size. The SLAB allocators, on the other hand, reserve a number of pages for a SLAB cache, and can then hand out objects of a smaller size from that cache. Each cache contains objects of a single size. Most OS kernels contain a similar SLAB allocator, including Linux, Solaris, FreeBSD, and NetBSD.

To provide variation in these dynamic allocators, we make sure one variant will follow different allocation patterns than the other. In particular, for the `vmalloc` allocator, we simply add guard pages around each allocation. Since `vmalloc` is rarely used and its allocations are not physically contiguous, this adds minimal overhead.

The pressure and constraints on the SLAB allocator are much higher, and therefore it requires a different scheme to provide diversity. In our design we change one variant to have a type-based SLAB allocator. While Linux partially provides this interface, allocations made with `kmalloc` still use (untyped) generic caches. Furthermore, Linux by default merges all caches with the same size. By making the SLAB allocator type-based, we not only get type-safety, but we also get different allocation patterns for every cache, since each data structure now has its own cache.

Stack frames Since a lot of info leaks originate from the kernel stack it is important to provide variation there as well. The layout of the stack is determined by the compiler, and cannot be changed after compilation. We identify two types of info leaks on the stack: an overflow into another variable on the stack or an uninitialized read on the stack (Fig. 5.2a). For each of these we propose lightweight variation techniques using a compiler pass, leading to a different layout, causing different behavior for both classes of errors.

To protect against out-of-bound reads on the stack we change the kernel stack frame layout: by modifying the order or size of every stack variable an over-read will end up reading a different variable in each variant. For uninitialized reads, the read will instead read contents that were on the stack before from an old stack frame, as shown in Figure 5.2a. These contents were placed there by a previous function call, from another path in the call-chain. To protect against such errors, we randomize the distance between stack frames as shown in Figure 5.2b. In each variant the stack frame of the function containing the uninitialized read will end

up in a different location. Moreover, its offset relative to the old stack frame (`func2` in Figure 5.2) will be different. While both variants might read sensitive data on their own, the chance that they both read the same sensitive data is marginal.

5.5 Implementation

As previously discussed, kMVX requires four building blocks: 1) multiple kernel instances, 2) variant generation, 3) syscall sync, and 4) I/O sync. In this section we will explain in detail how we modified the Linux kernel to implement these.

Popular operating systems capable of running **multiple kernel instances** include Barrelfish [15] and Amoeba [147]. For compatibility with existing software and to show that our design can be applied to production-use operating systems, we implemented a prototype of the kMVX design for Linux, on top of FT-Linux [127]. FT-Linux allows multiple Linux kernel instances to run on the same multicore machine for the purpose of fault tolerance, but provides none of the variant generation and synchronization functionalities part of our kMVX design. FT-Linux is based on the Linux 3.2.12 kernel and targets the Intel x86-64 architecture. We firmly believe that our design can be applied to other OS kernels and enabled on other ISAs as well.

In kMVX the hardware resources of the machine are split into two partitions: each kernel instance gets a set of cores and its own reserved memory. We run all unprivileged user space processes in a special kernel namespace with syscall and I/O sync enabled. When a process is started in this namespace, the same process is also created on the follower, having the same PID, file descriptors, and environment.

Contrary to the design shown in Figure 5.1, in our prototype each kernel runs a separate version of the user program for each kernel variant. Most system processes run only once on a machine: only unprivileged applications run on both variants. Running not only the kernel but also applications twice is easier to support, requiring less communication between the kernels, and most importantly eliminates sharing of possibly critical data structures between variants. By strictly separating the memory we reduce the possibility of circumventing kMVX by leaking shared data. In other words, running two separate versions of the user program introduces more variance in the memory layout of the kernel and allows the variants to have strictly separated memory. Similarly, by running the application on the follower, we also introduce variance with scheduling, which may be a source of bugs (e.g., race conditions). Note that replicating the user space application is an implementation detail that can be changed, as it is not fundamental to our design.

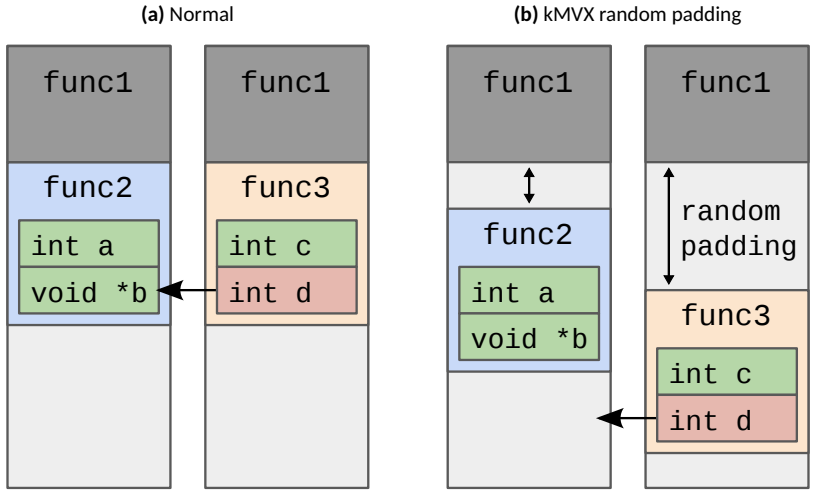


Figure 5.2: The layout of the stack with different stack frames per function (stack grows downward). This shows the effect of uninitialized reads with and without stack frame padding. In both cases `func1` first calls `func2`, then `func3`. In `func3` there is an uninitialized read on the variable `d`, which would read the value that was previously there, the pointer `b`, but with random padding between frames the value will differ.

5.5.1 Variant generation

Address space partitioning We constructed an address space partitioning mechanism, which allows us to linearly split the physical memory space of one machine between different kernels. This feature implicitly enables the Linux kernel to deflate and load its image at any physical address. We can freely set these addresses with kernel boot parameters.

For the virtual addresses, the Linux kernel divides its address space into several regions (direct physical mappings, vmalloc, virtual memory map, kernel text, and modules). To implement address space partitioning, we logically halved the size of each virtual region and assigned the first half to the leader and the second half to the follower, thus keeping the original base address for the leader, and assigned a new one to the follower, as shown in Figure 5.3b. This has the side effect of reducing the maximum amount of supported physical memory to 32 TB. For this purpose, we had to slightly modify the kernel to ensure each memory region could be freely relocated in the virtual address space. On recent Linux versions, most memory regions are subject to kASLR already, enforcing the memory regions to be in a predefined order and size. We modified the kernel to ensure physical-to-virtual offsets, order, and size can be freely changed and randomized at compile time for all the virtual memory regions, addressing the relocation problem to support address space partitioning.

Stack and heap variation Besides *address space partitioning*, we implemented all vargen techniques described in Section 5.4.3 for our prototype, providing us with a high level of security against various classes of vulnerabilities. Note that we can defend against more classes of vulnerabilities by adding more variation techniques. In this chapter we focus on a few stack- and heap variation techniques. Other memory layout randomization techniques, such as struct field order randomization [78], have successfully been applied to Linux [193].

For our prototype we modified the SLUB allocator (used for `kmalloc` and `kmem_cache_alloc`) to be type-based. For the untyped `kmalloc` allocator, we assign types based on the call site, looking at the return addresses as unique type signature [5, 201]. For dynamically sized objects such as strings, we still use the old `kmalloc` (size-based) caches. Changing the allocator introduces variance in the heap layout, providing (probabilistic) protection against heap-based vulnerabilities, including use-after-free vulnerabilities.

For the stack variation we modified the GCC compiler (version 6.0) to add random padding to stack frames, and reverse the order of variables within stack frames. These variations have minimal impact on performance, while providing kMVX with strong protection against stack-based info leaks, as detailed in Section 5.4.3.

5.5.2 Syscall sync

Our kMVX prototype synchronizes on every `copy_to_user` and `put_user` call. When the leader encounters a synchronization point, the contents of the copied data is sent from the primary to the follower using a fast inter-kernel communication channel. Each message sent between the variants is given a unique id, based on the current process id, syscall id, and a message counter, allowing us to match a certain sync call that originates from the same syscalls on both variants.

Upon entering the `copy_to_user`-call, the follower reads the data from the channel and does a byte-by-byte comparison of the buffers. If the expected message is not available in the channel, to avoid busy waiting, we let the kernel schedule another kernel thread when waiting for a message. We made the scheduler kMVX-aware, allowing it to prioritize tasks with available sync messages. If the follower does not detect a divergence in the message, it sends the leader an acknowledgement message, allowing both variants to continue execution. If, however, the follower detects a divergence, it sets the offending bytes to a zero-value, logs the divergence, and sends the modified buffer to the leader. The leader, finally copies either its own buffer in case of an acknowledgement or the modified buffer in case of a divergence.

In our experiments, zeroing the data does not cause any conflicts and allows us to prevent possible info leaks without halting the user-space application. Also note

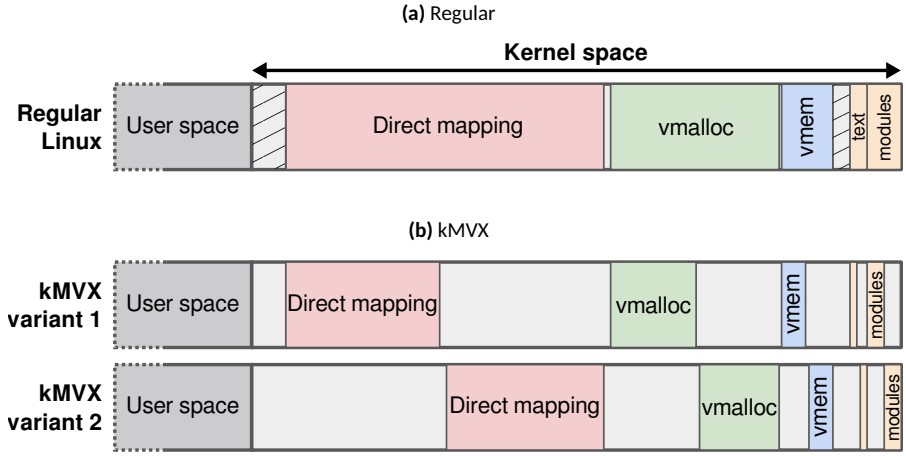


Figure 5.3: Regular kernel virtual address space, and the kMVX partitioned non-overlapping address space.

that zeroing the data may only cause a conflict affecting the user space application; the kernels do not diverge or crash due to a mismatch between the leader and follower.

I/O-intensive workloads might generate a lot of communication between variants, both for I/O- and syscall sync, potentially slowing down kMVX. Hence, it is vital to implement an efficient communication channel for our *sync points*. Since the messages for kMVX are synchronous in nature, we opted for an efficient communication layer based on a lockless shared-memory hash table. We also implemented a shared-memory allocator, to reduce the number of memory writes when transferring messages between the variants.

5.5.3 I/O sync

Kernel operations that interact with hardware devices cannot be performed by both kernels simultaneously. In cases such as *network I/O*, we let the leader communicate with the hardware, after which the result is *replayed* to the follower. This approach is comparable to replaying non-idempotent syscalls in existing MVX systems, but applied to operations inside the kernel. When the leader executes these replayed calls, it copies the results to a shared buffer from where the follower can read the result using a similar communication channel as *syscall sync*. Note that contrary to syscall sync, I/O sync does *not need to be synchronous*; the leader does not need to wait for an acknowledgement from the follower.

kMVX currently contains two different I/O sync modes: *low-level* and *high-level* sync. In the *low-level* sync component kMVX replays results of `in(b,w,l)`, `read(b,w,l,q)`, and `memcpy_fromio` calls, which are used to interact with hard-

ware. This *low-level* mechanism is not used for all drivers, only the ones whitelisted by kMVX. By synchronizing these reads we maintain a consitent state for simple devices such as the real-time clock without special modifications. More complex device I/O, such as DMA, cannot be transparently supported currently, requiring modification in drivers to be kMVX-aware. For compatibility reasons, these kinds of I/O effects are replayed at a *higher* (i.e., subsystem) level. kMVX provides a wrapper mechanism which makes it easy to replay subsystem calls with a few lines of modifications.

Networking Since the networking stack is highly state-dependent it requires heavy modifications to keep synchronized. For most of the networking, in our prototype, we opt to replicate the calls at a higher level, manually marshalling the effects of the syscall from the leader to the follower, while keeping a minimal shared state to reduce communication.

For example, to keep the result of the `select` call consistent across variants, we have to replay the resulting sets of the select call from the leader to the follower. Nevertheless, we leave most error handling up to the individual variants. For example checking for invalid file descriptors is left to the individual variants. This approach gives us a balance between full state-replication and just forwarding the results of the syscall from the leader to the follower.

As another example, in the `socket` syscall we synchronize the sets of open file descriptors between variants, mainly to keep the file descriptors consistent, but also to allow for error handling in the follower without communicating with the leader. When the `read` syscall is called on a socket file descriptor, the leader has to replay the data obtained from the lower levels of the network stack to the follower, thus requiring an I/O sync. On the other hand, if `read` is called on an invalid file descriptor, there is no need for an I/O sync, as both variants know about which file descriptors are valid.

In the future I/O sync can be changed to happen at a lower level in the networking stack, but this requires more complex logic to be kept consistent. Currently our prototype only supports the `select` API for network I/O multiplexing. As such, we do not support replaying of the `poll` and `epoll` interface, however, these interfaces could be added in a similar fashion to the `select` syscall.

Disk I/O The current implementation runs different file systems for each replica. The leader kernel accesses the physical disk, while the replica uses a `tmpfs` file system. Using different drivers is a source of variance, enhancing security as the execution paths are very different. Since copy-to-user calls are implemented at a higher level (i.e., not in lower-level drivers) the output is still the same. For kMVX we implemented a generic *character device* that can replay an arbitrary device. For example, to eliminate divergence in the variants, we apply this character device to

`/dev/(u)random`, replaying the value from the leader to the follower.

Furthermore, a number of syscalls may cause divergence between variants, due to nondeterministic behavior. By replaying these syscalls from the leader to the follower, we keep the variants consistent. For example syscalls which may return different values if the variants are even slightly out of sync, such as `gettimeofday`, are replayed from the leader to the follower, while syscalls that do not need to be replayed are executed locally on both variants. In fact, only a *small subset* of all syscalls are replayed. For example, we modify the `getpid` syscall to return a namespace-unique identifier that is the same across replicated processes in the variants, avoiding extra communication overhead. Our current prototype replays syscalls related to *time* (`time` and `gettimeofday`) and *networking* (in total 15 syscalls were modified for replaying).

Nondeterminism A source of nondeterminism in the kernel, besides hardware I/O, is kernel thread scheduling. Since the scheduling order inside the kernel is not visible to user space, it does not affect kMVX. The order of system calls in user space might, in the case of multithreaded applications, cause divergence. kMVX supports deterministic Pthread replication by forcing a global deterministic order on the system calls for a process. By using a modified Pthreads library (loaded using `LD_PRELOAD`) that overrides functions, such as `mutex_lock` and `mutex_trylock`, enabling deterministic ordering of syscalls through a special system call. We refer the reader to the FT-Linux paper [127] for a more thorough explanation of thread replication.

Likewise, interrupts are another source of nondeterminism. While these sources of nondeterminism affect the control-flow in the kernel [157], the interaction with user space remains unchanged. The only exception being signals. In order to keep the variants consistent, signals caused by the user (e.g., using the `kill` syscall) are delayed until after the next synchronization point. Likewise, signals originating from other processes are also delayed is a similar fashion.

Asynchronous signals originating from interrupts, such as `SIGINT` caused by the keyboard, are a source of randomness, as they can be caused at any moment in the execution. In our prototype we delay dispatching this signal until the next syscall, as to have a coherent state between variants. We then initiate the same signal on the follower using a callback function through an asynchronous messaging layer. Handling asynchronous signals in MVX is an open problem, with several solutions having been proposed [179, 180]. Our standard `put_user()` and `copy_to_user()` modifications check signal data copied to user space for consistency across variants. We had to modify `copy_siginfo_to_user()` to not synchronize pointer fields (which will obviously be different across variants due to our variation techniques).

5.6 Evaluation

5.6.1 Performance evaluation

Experimental setup We evaluate the performance of our kMVX prototype using a number of *micro* benchmarks as well as a number of real-world application *macro* benchmarks on a 4-core Intel i7-3770 with 16 GB of RAM. All benchmarks compare the performance of kMVX against a vanilla Linux 3.2.12 kernel. When running kMVX, we split the hardware into two partitions: each variant is limited to *2 cores* and *8 GB of RAM*. This partitioning accurately reflects how deploying kMVX on existing systems affects the performance without adding additional resources. The baseline uses *4 cores* and *16 GB of RAM*. We disable the vDSO for both the baseline and for the kMVX kernels. For the real-world server applications we benchmark the in-memory database redis-4.0.6 and three web servers: nginx-1.10.1, lighttpd-1.4.48, and a mongoose-6.10-based web server. We also benchmark the performance of multithreading in kMVX using the parallel compression utility *pbzip2*. Since our prototype implementation runs two kernels and two copies of the user space application, kMVX uses roughly twice as much memory as a vanilla Linux kernel.

Microbenchmarks Figure 5.4 presents the overhead incurred by our kMVX prototype by comparing the median of the number of cycles taken per syscall measured from user space relative to a vanilla Linux kernel. Each of the selected syscalls is executed 10,000 times in a loop.

In Figure 5.4, the overhead of the microbenchmarks is split into three components: overhead of the normal execution of the syscall, the I/O sync, and the syscall

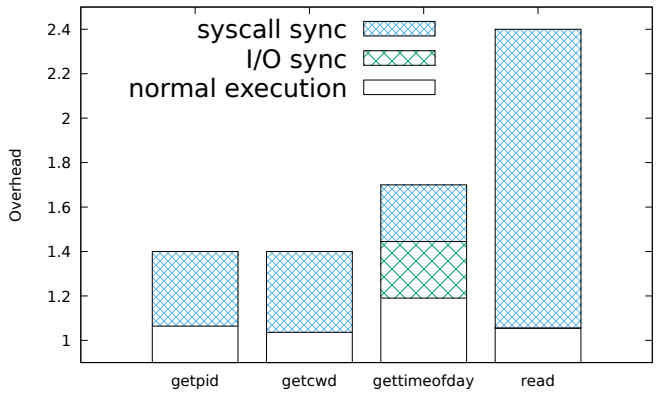


Figure 5.4: Microbenchmarks. Relative performance overhead of a selected number of syscalls. For the read syscall we read a 512 KB file.

	Baseline	kMVX		UniSan	KASAN
	μs	μs			
null call	0.04	0.06	(50.0%)	0%	5%
null I/O	0.08	0.14	(75.0%)	–[b]	49%
stat	0.34	0.45	(32.4%)	2.7%	640%
open/ close	0.73	0.92	(26.0%)	-4.2%	1300%
select TCP	2.06	2.47	(19.9%)	0%	59%
signal install	0.11	0.15	(36.4%)	0%	10%
signal handle	0.74	2.41	(226%)	3.4%	311%
fork proc	67.1	101.31	(50.1%)	0%	299%
exec proc	204.0	275.11	(34.9%)	0.7%	163%
sh proc	483.0	737.60	(52.7%)	–[b]	50%
pipe latency	1.77	2.05	(15.8%)	2.4%	41%
prot fault	0.208	0.209[a]	(0.4%)	2.4%	26%
TCP latency	12.0	21.10	(75.8%)	6%	250%
	MB/s	MB/s			
Pipe bw	3005	2100	(30.1%)	0.2%	27%
TCP bw	3330	1641	(50.7%)	-0.1%	60%

[a] At the moment our prototype does not accurately synchronize all signal handling events between the kernels. To prevent false positives, we disable kMVX sync for asynchronous signals.
[b] Not reported in [128].

Table 5.1: LMBench results for kMVX

sync. The syscall sync includes the overhead for communicating return values to other variants, waiting for the other variant, and checking if they match, whereas I/O sync overhead is the cost of replaying certain logic in the follower.

As shown in Figure 5.4, simple syscalls such as `getpid` and `getcwd`, which copy a small amount of data to user space, incur a overhead of about 40%. The overhead mostly comes from *syscall sync* between kernels. While `gettimeofday` also copies a small amount of data, this syscall needs to be replayed from the leader to the follower (I/O sync), in order to keep the view of time between the variants consistent to prevent spurious divergences. On the other hand, the `read` syscall returns a large buffer to user space via copy-to-user. kMVX splits larger buffers into several cache-aligned blocks to optimize throughput. In Figure 5.4 we can see that the `read` system call has a higher overhead than other syscalls due to the larger buffers being copied for syscall sync.

We also benchmark kMVX using the LMBench suite of microbenchmarks, as presented in Table 5.1. Results show that the overhead is generally more prominent for I/O-intensive benchmarks. In general, small syscalls with a low duration have a higher overhead, since a relatively larger part of the syscall is spent on syscall sync. The select TCP benchmark has a lower overhead compared to other benchmarks due to the fact that it is replayed, thus the leader does not need to wait for an acknowledgment from the follower. Note that LMBench stresses the system calls

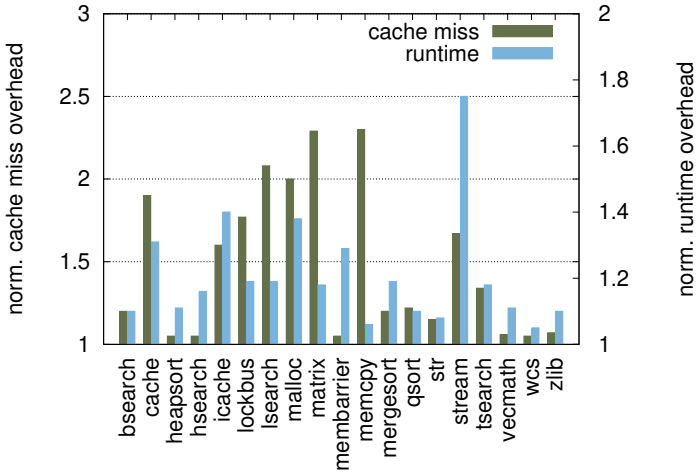


Figure 5.5: stress-ng. We measure the performance of kMVX on user space benchmarks making use of few syscalls. We show how cache performance is affected by kMVX. *stress-ng* is configured to use two workers.

(where a large part of the overhead for kMVX comes from). As such, the overhead is higher than for a majority of real-world applications, and shows the worst-case for kMVX.

Since running multiple kernels introduces more memory accesses, we also benchmark how kMVX affects cache hit/ miss rates compared to a vanilla Linux kernel. For this purpose, we use the *stress-ng* [194] benchmarking suite to measure cache performance and to show how kMVX behaves for computationally and memory intensive applications (i.e., applications with relatively few syscalls).

As can be seen in Fig. 5.5, the cache miss rate is significantly higher for kMVX than for a vanilla Linux kernel. For applications that are computationally heavy, rather than memory heavy, we generally see a modest 22% runtime overhead. For more memory intensive applications, we see that the overhead is proportional to the increase in cache miss rates in most cases. For example, the `cache`, `matrix`, and `stream` benchmarks in the *stress-ng* suite are memory intensive, getting a higher cache miss rate when running on kMVX, degrading their runtime performance. Some benchmarks (such as `icache`, `malloc`, and `membarrier`) perform many syscalls, and as such kMVX takes an significant performance hit, as shown in earlier microbenchmarks.

Some benchmarks, such as `zlib`, `wcs`, `vecmath`, see a modest increase in cache misses. The data accessed by these benchmarks usually is available in L2, giving us a cache hit. Since L2 cache is partitioned per-core, and the cores are not shared between variants, we speculate that the L2 miss rate is not impacted as much in

these kinds of benchmarks. On the other hand, the `stream` benchmark shows a significant higher cache miss rate. This benchmarks allocates buffers at least 4 times the size of the L2 cache, and repeatedly performs operations on these buffers. The `stream` benchmark, thus relies on L3 cache, which is shared by both kernels in our setup, showing a much higher cache miss rate than other benchmarks. Architectures with non-inclusive caches may show different characteristics for these benchmarks.

Macrobenchmarks To evaluate the real-world overhead of our kMVX prototype, we benchmark a number of server applications. Servers incur the most overhead from kMVX since they are particularly I/O-intensive, their primary bottleneck being networking or the disk. Syscalls like `read` occur frequently in such applications and require expensive checks on the copy-to-user calls, as shown by our microbenchmarks. Furthermore, network syscalls have to be replayed, adding more overhead for such applications.

Figure 5.6 shows the overheads for the *mongoose*, *nginx*, and *lighttpd* web servers for a varying number of concurrent connections. We benchmark kMVX using ApacheBench (with keepalive flag, loading a 512 byte page) over a 1 gigabit network connection (the client running a Intel i7-7800X with 24GB RAM), however, in this setup we could not saturate the system fully for some benchmarks. For example, as can be seen in Figure 5.6 when using a 1 gigabit network connection the overhead for *mongoose* becomes 3%, as the network request latency is the major bottleneck for the single-threaded server. For *nginx* and *lighttpd* we see an overhead of at most 27% on the gigabit connection.

To be able to fully saturate the system, we also benchmark using the loopback interface, achieving the highest possible throughput. In this setup ApacheBench itself only runs on the leader kernel, whereas the server application utilizes both variants. Since using the loopback interface eliminates any overhead from the kernel drivers this represents a worst-case scenario. The overhead via the loopback interface for *mongoose* is about 35% when saturated (i.e., more than 10 concurrent connections). For the better-performing servers, *nginx* and *lighttpd*, we see a higher overhead of about 50% when saturated (around 20 concurrent connections).

Similarly, the average runtime performance overhead is 43% when we stress the server with the `redis-benchmark` suite (Figure 5.7). In some parts of the suite, such as the `SET` and `LPOP` tests, the performance overhead is in the range of 26%. The relatively better performance for these tests can be explained by the fact that a larger part of these operations are performed in user space. The impact of syscall sync on the memory-intensive `SET` operation is relatively smaller than for other syscall-bound operations, such as `PING`.

Besides I/O heavy applications, such as the servers discussed earlier, we also

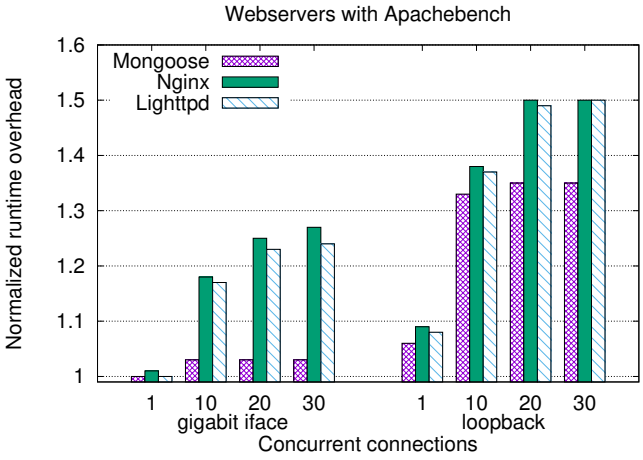


Figure 5.6: Web servers benchmarked with ApacheBench on a 1 gigabit link and the loopback interface. All web servers use the `select` API. Nginx has 1 worker process.

show the performance of kMVX on the multithreaded *pbzip2*, which is computationally/ memory-heavy, rather than I/O-heavy. In this benchmark, we compare the data throughput of kMVX against vanilla Linux in a *multithreaded* file compression benchmark. *pbzip2* is an application that uses *Pthreads* for parallel file compression. pbzip2 uses one producer thread, reading the file from the disk; a number of *worker* threads to compress the file in parallel; and one consumer thread, combining the output from the worker threads into one file. In Fig. 5.8, we see that the throughput in kMVX when using 1 worker thread is about 11% lower than for a vanilla Linux kernel. A large part of this degradation is due to the overhead incurred by the syscall sync. In the case of *pbzip2*, there are no significant syscalls in the run of the program besides the `read`, `write` and `futex` syscalls. Enforcing a total order on the syscalls in the various threads is, thus, relatively cheap.

When partitioning the hardware for kMVX we give up half of the cores usually available to user space applications. As such, the performance with more than 4 worker threads is, naturally, lower for kMVX. If we consider only the performance up to 4 worker threads, the geometric mean of the run-time overhead is 16%. We note a slight decrease in performance when using more threads in kMVX, caused by the overhead incurred by keeping a global order on the system calls over the various threads.

Note that the overhead of 16% (when considering up to 4 workers) is significantly lower than the $30\%+$ observed for syscall-intensive applications, such as servers. Most of the time in *pbzip2* is spent in user space, making the syscall sync have less of an impact on the overall performance.

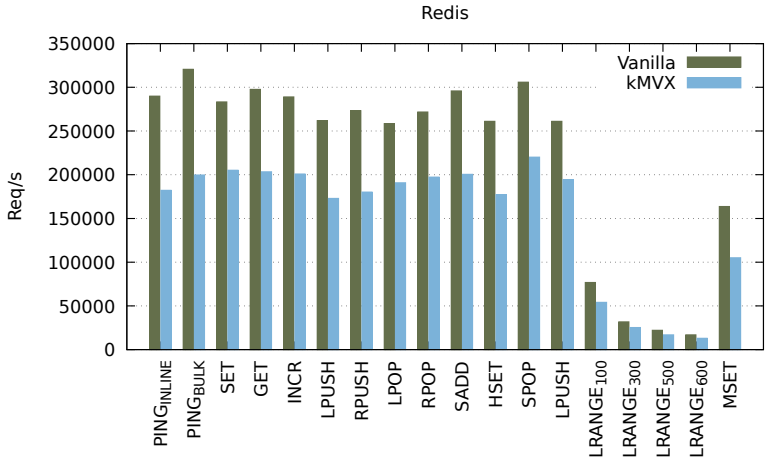


Figure 5.7: Redis. Geomean overhead of ∼1.43x for the *redis* in-memory database using redis-benchmark.

Comparison with related work Our prototype shows an overhead similar to that of traditional user space MVX systems. It is, however, a debatable comparison, since kMVX operates at the kernel level. Instead, we compare our prototype to other kernel defenses focusing on information leaks. For instance, KASAN and kmemcheck provide some degree of memory safety in the Linux kernel, but do not cover all classes of info leaks and have respectively 4x and multiple orders of magnitude overhead. Recent publications on mitigating uninitialized read vulnerabilities, UniSan [128] and SafeInit [139], are the closest to kMVX. Note that UniSan and SafeInit only prevent leaks for uninitialized data. kMVX presents a more general defense against information leaks, both for uninitialized data and for other vulnerabilities, such as buffer overreads of initialized data. We would like to point out that the approach introduced with kMVX can also be applied to prevent other types of exploits besides information leaks. For example, MVX has previously been applied to preventing ROP-based attacks [207]. In this chapter, our variation techniques are focused on preventing *information leaks*.

Existing solutions for preventing information leaks in the kernel, focusing only on uninitialized reads, have a very low performance overhead. SafeInit and UniSan have an average overhead of below 5% for typical server applications. Note that UniSan benchmarks their web server on a 1 gigabit connection. In the same setup kMVX has a overhead of at most 27%. As a comparison, we included the *LMBench* numbers from the UniSan paper in Table 5.1. While the UniSan numbers show much less overhead than kMVX, UniSan can only prevent uninitialized reads, which constitute only 60% of all info leaks [128]. kMVX provides *stronger security*

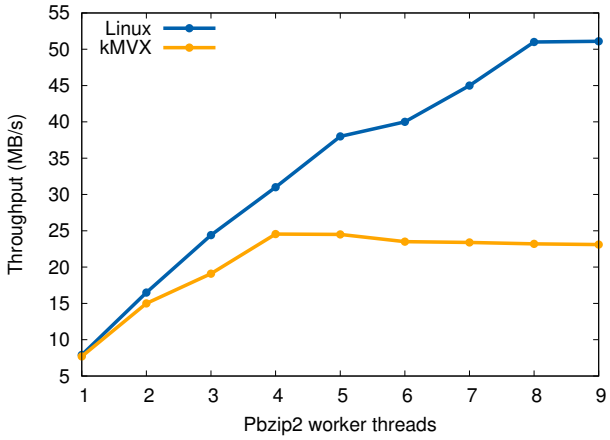


Figure 5.8: pbzip2. Benchmark of multithreaded bzip2 compression of a 1 GB file of random data using different number of worker threads, using the default block size of 900K. We show the throughput in MB/s of vanilla Linux and kMVX. Note: when kMVX is enabled, we only have 2 cores available.

guarantees by covering other classes of info leaks not detectable by previous defenses. On the other end of the spectrum, regarding memory safety in the kernel, we find KASAN [123]. KASAN offers detection of various memory errors, such as buffer overreads, use-after-free, use-after-return. Note that KASAN is intended as a debugging tool for memory errors, rather than a defense mechanism against information leaks, and as such the overhead is, understandingly, significant.

5.6.2 Security analysis

For our security analysis, we identify two primary targets that attackers want to leak from the kernel: kernel *pointers* for further attacks and *sensitive data* such as crypto keys. kMVX applies address space partitioning to its variants, meaning *every* pointer will differ between variants. When a valid kernel pointer is copied to user space, the corresponding pointer will be different in the other variant, leading to a *detected divergence*.

For other sensitive data, we have to consider where it is stored and leaked from. For instance, such data might be located on the kernel stack or heap, and in each case the guarantees depend on the variation in those areas.

For the *heap*, our design provides strong guarantees against both spatial and temporal memory error exploits. Because of the type-based SLAB allocator, both temporal and spatial attacks are already more limited [5, 201]. Having allocators with different behaviors and allocation patterns between variants is highly likely to eliminate the residual attack surface. For the stack, data can be leaked via temporal

Type of leak	# CVEs			kMVX	UniSan	KASAN
	H	S	O			
use-after-free	15	0	0	15	0	15
uninitialized read	49	4	1	54	54	0
out-of-bounds read	1	1	0	1	0	1
other	8	0	10	12	0	0
Total		89		82	54	16

Table 5.2: Reported Linux kernel 2017 information disclosure CVEs categorized by type and origin of leaked information.
H: heap, S: stack, O: other.

issues, such as an uninitialized read, or a spatial error such as a buffer overflow. The former is stopped by having different reuse patterns for each variant, using random stack frame padding. Spatial errors are stopped by the stack frame padding or the variable order randomization, causing data to be misaligned in both variants, triggering divergence and detection.

To concretely demonstrate the effectiveness of kMVX, we analyzed all the Linux kernel information disclosure vulnerabilities (CVEs) from 2017 [162]. Similarly to prior work in the area [128, 171], we split the types of information leak vulnerabilities into four categories: (1) *uninitialized read*, (2) *use-after-free*, (3) *out-of-bounds read*, and (4) *other*. The *other* category contains vulnerabilities that leak information using unconventional means, either through a faulty check or by writing sensitive information to a location accessible by an unprivileged user (e.g., system logs). To show that our variation techniques are sufficient, we also identify the origin of the leaked data for each vulnerability: heap (*H*), stack (*S*), and other (*O*). Our analysis shows that our variation techniques are sufficient in preventing 92% of info leak CVEs published in 2017. An overview of the analysis of all these CVEs is published as part of the source code of kMVX.

To showcase how kMVX stops vulnerabilities, we randomly picked three CVEs concerning info leaks in the Linux kernel. *CVE-2016-4569* leaks information from the kernel *stack* due to *reading uninitialized data*. In our tests, we were able to reliably detect that information was leaked as our stack variation techniques between the variants make the diversified stacks leak different data. Similarly, due to the variations introduced by the different heap allocator implementations, *CVE-2013-2237*, which leaks information from the kernel *heap* using uninitialized reads, is also detected. In short, kMVX can reliably detect both heap- and stack-based uninitialized reads, as well as *out-of-bounds reads*, due to the diversified memory layouts introduced by our vargen.

In *CVE-2016-0728*, an attacker can exploit an *use-after-free* bug to forge an object at the location of a previously freed object. Since these objects are of different

types, the type-based SLAB allocator in the follower places this object in a different bucket, leading to detectable divergent behavior between the variants.

Finally, we also prevent *other* possible leaks of kernel pointers that do not originate from memory errors, but instead are accidentally leaked due to programmer errors. For example, until recently, the `wchan` field of `/proc/PID/stat` could be used to leak an absolute kernel address to unprivileged user space [145]. When an attacker reads such a file, kMVX will detect the divergence and zero out the differences between the variants, eliminating such info leaks.

The one out-of-bounds read vulnerability from the 2017 info leak CVEs (see Table 5.2) that kMVX does not detect is due to a remote attacker leaking information using a bug in a USB driver. As we focus on a local attacker in our threat model, we currently do not cover this vulnerability. The remaining 2017 CVEs that kMVX is not able to detect are either present in the *bootloader* (e.g., CVE-2017-0455) or are caused by timing side channels (e.g., CVE-2015-2877).

We have shown that kMVX can detect and prevent numerous types of vulnerabilities due to our selected variation techniques. By adding more variation techniques, other types of vulnerabilities can be thwarted in the future. To our knowledge, kMVX is the *first* defense to deterministically prevent any leakage of *kernel pointers* and provide strong (probabilistic) guarantees for preventing leakage of non-pointer data.

5.7 Related work

Kernel defenses Since Linux is a complex and integral part of the software stack, many kernel defenses have been developed over the years. For instance, kmemcheck [121] provides comprehensive memory safety (use-after-free, out-of-bounds, and uninitialized reads) but at the cost of orders of magnitude overhead. KASAN [123] focuses on only use-after-free and out-of-bounds bugs by applying ASan [186] to the kernel, but cannot detect all information leaks and still incurs 4x overhead. The use of annotations and static analysis with Coccinelle [120], Sparse [122], and GCC plugins [168] can detect potentially problematic code patterns statically. The PaX team introduced features to, for instance, force zero-initialization of `__user` structs (STRUCTLEAK), stack frame clearing during syscalls (STACKLEAK), harden `copy_to_user` (USERCOPY), and secure deallocation (SANITIZE) [167].

UniSan [128] and SafeInit [139] can prevent kernel info leaks due to reading uninitialized variables, by using static analysis and forcing initialization of variables that might leak to the user. Most of these approaches only address a single source of info leaks or incur significant runtime overhead, whereas our kMVX design addresses all info leaks at a much smaller cost. The SPLIT KERNEL [112] hardens the kernel with several (expensive) hardening techniques, but also maintains

non-hardened copies of each function, allowing an untrusted process to run under the hardened kernel whereas trusted processes can still use the more efficient kernel functions.

MVX In 2006, Cox *et al.* [52] proposed a design, based on N-version programming [40], where variants were automatically generated, and where the monitoring and synchronization happens at the syscall level, laying the foundation of modern MVX research. Variant generation is based on the field of (automated) software diversity [117]. Proposed variant generation include random code insertion [75], system call randomization [45], instruction set tagging [52], address space partitioning [52], heap- and stack layout randomization [20, 180], and non-overlapping offset spaces [107]. Our variant generation for kMVX takes inspiration from these and implements effective and efficient techniques suited for the kernel.

Monitoring for user space MVX is done primarily through the syscall interface, with older solutions using more coarse-grained alternatives [20, 52]. Three approaches exist: in-kernel monitoring (high performance but intrusive) [52, 208], monitoring by an external application (for instance using the `ptrace` debugging facility of the kernel, which is safe but slow) [33, 105, 180, 209] and in-process monitoring, which is efficient but requires a careful design in security applications [91, 115, 173, 174, 208]. An exception here is Detile [77], which instead synchronizes at the byte-code instruction level to protect scripting engines. Our kMVX design also works at a different boundary than syscalls, requiring a new design. It can be compared with both the in-kernel and in-process monitoring, and can be considered more intrusive than traditional MVX monitors.

5.8 Conclusion

In this chapter, we have presented kMVX—the first design that runs multiple diversified kernel variants simultaneously on the same machine to prevent information leaks. We have discussed a prototype implementation of kMVX, which provides efficient info-leak detection while preserving the Linux ABI, allowing us to run regular Linux binaries without changes or recompilation. We have demonstrated a number of variation techniques specific to kernels and identified the main obstacles in keeping the variants from diverging. We have also shown our prototype supports a number of popular server applications with a overhead of 20%–50%. On less I/O-intensive applications, such as *Pbzip2* and the *stress-ng* benchmark, the overhead is generally below 20%. The source code for kMVX is freely available at `https://github.com/vusec/kmvx`.

Contribution

This chapter was previous published as a research paper in collaboration with other authors. This paper was written in joint-effort with Sebastian Österlund, with whom I share first authorship. I was primarily responsible for the initial kMVX design and implementing variant generation, Antonio Barbalace did the initial porting of FT-Linux to support MVX, and Sebastian implemented most in-kernel synchronization and did all evaluation of the paper.

6 | Conclusion

After over 30 years of developments and research, memory errors still rank amongst the biggest computer security issues. Mitigating such issues automatically and transparently to the programmer is difficult, and all attempts to harden existing C and C++ code suffer from trade-offs in compatibility, performance, and/or security. In this dissertation we looked at different new insights in providing memory safety. In particular, we demonstrated that existing, commodity hardware features can be used or repurposed for improving memory safety, and derived the following insights in various areas of memory safety research:

Protecting safe regions We analyzed various alternative memory safety solutions. Such systems can often easily be bypassed by compromising their metadata regions, because these sensitive regions are not properly protected. Where it was previously believed that protecting such regions is too costly, we demonstrated in Chapter 2 multiple options do exist. For example, protecting a shadow stack can be done efficiently using address-based isolation, costing only 4% runtime overhead using traditional SFI, and 2.8% overhead when repurposing the Intel MPX extensions for SFI. This demonstrates Intel MPX has a practical use case, despite being deprecated for its original purpose of bounds checking [86, 116]. For less frequent accesses to the secure metadata, Intel MPK is ideal. But for systems lacking this recent feature we show that other methods, such as virtualization extensions, are a viable alternative. Since the time of this research, Intel has released processors supporting its MPK extensions, which show overhead comparable to the ones presented in Chapter 2.

Bounds checking and pointer tagging We presented a unique design for a bounds checker, that implicitly updates its metadata and automatically delegates its checks to the (unmodified) MMU. Through this design, and by focussing on the common case of buffer overflows (and not underflows), we reported an overhead of only

35% where the next best solution has over 60% (for under- and overflows). A major contribution of this work is its in-depth analysis of pointer tagging, especially for designs where the tag is not constant. We highlighted problems found in real-world code, and demonstrated solutions to make pointer tagging feasible without specialized hardware features.

Multi-variant execution performance and security With MvArmor, we presented a comprehensive defense for protecting programs where the source code is not available. Through a technique called MVX, we run multiple copies of the same binary simultaneously. By varying its environment (such as the address space layout and heap allocator) and comparing these variants at system calls, we can detect and prevent various forms of exploits. We presented a more comprehensive set of variant generation strategies, that better protect against memory errors than previous MVX solutions. Another contribution to the MVX field is our system call interposition design, where we leverage virtualization extensions to create light-weight VMs, which significantly speeds up MVX while still offering proper isolation.

Multi-variant execution for the kernel Finally, we presented the first design for applying MVX to an operating system kernel in our kMVX prototype. Our design tackles various problems, including how to run multiple variants of a kernel on the same machine, how and where to compare these variants, and what variant generation strategies can be used in the kernel. Our prototype builds on top of FT-Linux for running multiple kernels on the same machine, and extends it by synchronizing at several different interfaces that the kernel uses (system calls and I/O). Through a number of variation techniques, including modified kernel allocators and alternative stack layouts, we can offer comprehensive security against information leaks. Our synchronization components nullify any difference in data leaving the kernel, mitigating information leaks while maintaining availability of the system.

6.1 Future directions

Isolation of sensitive data We can already see an increase in proper protection of sensitive metadata for defenses, and we expect this will become the norm going forward [138, 200]. For example, Microsoft uses its hypervisor to protect the metadata of Control Flow Guard [138]. However, the increase of hardware-assisted memory defenses, such as Intel CET [94] and ARM MTE [8], will partially eliminate the need for explicit metadata isolation. Another trend that will likely continue is the interest in secure cloud environments, where customer data is isolated and protected from other tenants and the cloud vendor. Trusted execution environments such as Intel SGX and ARM TrustZone provide this support from the hardware. One problem that all aforementioned isolation techniques suffer from

is that of side channels, and more research towards ensuring the confidentiality of sensitive data is required.

Pointer tagging Our work on Delta Pointers demonstrated pointer tagging is a practical and efficient method for maintaining per-pointer metadata. While this idea has been around for decades [84], recent interest in efficient memory safety has caused a renewed interest in pointer tagging. With recent architectural additions to SPARC [160] and ARM [7], pointer tagging is even supported system-wide, eliminating the need for software masking. Such (minimal) extensions can be leveraged by new memory safety systems to increase both compatibility and performance, and we believe this is an interesting direction for future research.

Multi-variant execution optimizations MvArmor demonstrated an efficient design for monitoring MVX variants, through light-weight hardware-supported virtualization. We believe that such a design is amenable to further optimizations. The monitor in MvArmor currently forwards most system calls directly to the Linux kernel, resulting in overhead from the VM exit and further processing in the Linux kernel. We believe this design can be optimized by batching system calls [191] and implementing more I/O logic inside the monitor (which is effectively an libOS) to prevent VM exits altogether [19, 111, 172].

Multi-variant execution for kernels With kMVX, we presented the first design and prototype for applying MVX to operating system kernels. For our prototype, we focussed solely on information disclosure bugs, and did not fully support all I/O interfaces and drivers. Further research towards more complete coverage of all kernel interfaces can make kMVX more applicable. Furthermore, MVX in the past has been amenable to greatly diverse security guarantees, by plugging in different variant generation and synchronization policies. We expect this to also be the case for kMVX, where it can be used to protect against all types of different bugs.

Temporal safety This dissertation did not focus on ensuring temporal memory safety. One direction for new and more efficient temporal safety is the usage of light-weight virtualization (as shown MvArmor) to create safe allocators. Designs such as Oscar provide temporal safety by creating virtual aliases for each new object, but suffer from high overheads. Primary reasons for these overheads are the need to context switch to the kernel often, and the expensive metadata management the Linux kernel does for each virtual mapping. By running a process in a small VM, we can give the allocator direct access to the page tables, while ensuring safety and isolation through the second level page tables in the hypervisor. Preliminary results in this direction show such an approach has potential for significant performance benefits.

Memory tagging architectures One promising direction towards full memory safety is architectural support for memory tagging. SPARC ADI [160] and ARM MTE [8] show such designs are practical to integrate in modern processors. In such systems, every pointer and every chunk of memory has a tag (an n-bit identifier). On memory accesses, the hardware asserts there is a match between the pointer tag (encoded using pointer tagging) and memory tag (encoded in additional memory bits or a shadow map). To make implementations practical, only 15 different tag values are supported, reducing the security significantly. Further research in uses and optimizations for such tagged architectures show great promise [187], both on the hardware and on the software side.

Full memory safety on commodity hardware Providing memory safety for existing systems remains an open problem. State-of-the-art full memory safety systems are still impractical [35, 48, 149], and often-used solutions such as AddressSanitizer [186] are only useful for debugging, and cannot provide proper security. Many designs for spatial and temporal memory safety exist, but are generally not composable. We believe that, by taking inspiration from tagged architectures, a full memory safety system is possible without actually requiring customized hardware. HWAsan [187] demonstrates a prototype with $\sim$2x overhead, that only requires architectural pointer tagging support. Based on preliminary experiments, we believe this can be optimized further whilst eliminating the need for architectural pointer tagging altogether.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS*, 2005.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: principles, implementations, and applications. *ACM TISSEC*, 2009.

[3] Alessandro Acquisti, Allan Friedman, and Rahul Telang. Is there a cost to privacy breaches? An event study. *ICIS*, 2006.

[4] Sam Ainsworth and Timothy M. Jones. MarkUs: drop-in use-after-free prevention for low-level languages. In *S&P*, 2020.

[5] Periklis Akritidis. Cling: a memory allocator to mitigate dangling pointers. In *USENIX Security*, 2010.

[6] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy Bounds Checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security*, 2009.

[7] ARM. *ARM Cortex-A Series – Programmer's Guide for ARMv8-A*. 2015.

[8] ARM. Armv8.5-A Memory Tagging Extension. Technical report, 2019.

[9] Mike Ash. Friday Q&A 2013-09-27: ARM64 and You. `https://www.mikeash.com/pyblog/friday-qa-2013-09-27-arm64-and-you.html`, 2013.

[10] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, 1994.

[11] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[12] Algirdas Avižienis and Liming Chen. On the implementation of N-version programming for software fault tolerance during execution. In *COMPSAC*, 1977.

[13] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: real-time kernel protection from the ARM TrustZone secure world. In *CCS*, 2014.

[14] Michael Backes and Stefan Nürnberger. Oxymoron: making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security*, 2014.

[15] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: a new OS architecture for scalable multicore systems. In *SOSP*, 2009.

[16] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. Composing OS extensions safely and efficiently with Bascule. In *EuroSys*, 2013.

[17] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI*, 2014.

[18] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: safe user-level access to privileged CPU features. In *OSDI*, 2012.

[19] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: a protected dataplane operating system for high throughput and low latency. In *OSDI*, 2014.

[20] Emery D. Berger and Benjamin G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *PLDI*, 2006.

[21] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security*, 2005.

[22] Joe Bialek. Solving Uninitialized Kernel Pool Memory on Windows. `https://msrc-blog.microsoft.com/2020/07/02/solving-uninitialized-kernel-pool-memory-on-windows/`, July 2020.

[23] Joe Bialek. Solving Uninitialized Stack Memory on Windows. `https://msrc-blog.microsoft.com/2020/05/13/solving-uninitialized-stack-memory-on-windows/`, May 2020.

[24] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely rerandomization for mitigating memory disclosures. In *CCS*, 2015.

[25] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *S&P*, 2014.

[26] Hans Boehm. Bounding space usage of conservative garbage collectors. In *POPL*, 2002.

[27] Jeff Bonwick. The slab allocator: an object-caching kernel memory allocator. In *USENIX Summer*, 1994.

[28] Jeff Bonwick and Jonathan Adams. Magazines and Vmem: extending the slab allocator to many CPUs and arbitrary resources. In *USENIX ATC*, 2001.

[29] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: memory deduplication as an advanced exploitation vector. In *S&P*, 2016.

[30] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. Leakage-resilient layout randomization for mobile devices. In *NDSS*, 2016.

[31] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *VEE*, 2012.

[32] Marc Brunink, Martin Susskraut, and Christof Fetzer. Boundless memory allocations for memory safety and high availability. In *DSN*, 2011.

[33] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified process replicae for defeating memory error exploits. In *IPCCC*, 2007.

[34] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 2000.

[35] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CUP: comprehensive user-space protection for C/C++. *ASIACCS*, 2018.

[36] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: on the effectiveness of control-flow integrity. In *USENIX Security*, 2015.

[37] Nicholas Carlini and David Wagner. ROP is still dangerous: breaking modern defenses. In *USENIX Security*, 2014.

[38] Haibo Chen, Jieyun Chen, Wenbo Mao, and Fei Yan. Daonity - grid security from two levels of virtualization. *Inf. Secur. Tech. Rep.*, 12(3), 2007.

[39] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *ApSys*, 2011.

[40] Liming Chen and Algirdas Avižienis. N-version programming: a fault-tolerance approach to reliability of software operation. In *FTCS*, 1978.

[41] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security*, 2005.

[42] Xi Chen, Herbert Bos, and Cristiano Giuffrida. CodeArmor: virtualizing the code space to counter disclosure attacks. In *EuroS&P*, 2017.

[43] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. StackArmor: comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*, 2015.

[44] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, 2008.

[45] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical report, 2002.

[46] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the PDP-11: architectural support for a memory-safe C abstract machine. In *ASPLOS*, 2015.

[47] Clang Team. Clang 5 documentation: SafeStack. `http://clang.llvm.org/docs/SafeStack.html`, 2017.

[48] James Clause, Ioannis Doudalis, Alessandro Orso, and Milos Prvulovic. Effective memory protection using dynamic tainting. In *ASE*, 2007.

[49] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: minimal hardware extensions for strong software isolation. In *USENIX Security*, 2016.

[50] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security*, 1998.

[51] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *USENIX Security*, 2003.

[52] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: a secretless framework for security through diversity. In *USENIX Security*, 2006.

[53] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: practical code randomization resilient to memory disclosure. In *S&P*, 2015.

[54] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: a practical page-permissions-based scheme for thwarting dangling pointers. In *USENIX Security*, 2017.

[55] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *ASIACCS*, 2015.

[56] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. MoCFI: a framework to mitigate control-flow attacks on smartphones. In *NDSS*, 2012.

[57] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Snow, and Fabian Monrose. Isomeron: code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

[58] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security*, 2014.

[59] Liang Deng, Qingkai Zeng, and Yao Liu. ISboxing: an instruction substitution based data sandboxing for x86 untrusted libraries. In *IFIP SEC*, 2015.

[60] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[61] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE*, 2006.

[62] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *DSN*, 2006.

[63] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *PLDI*, 1998.

[64] Gregory J. Duck and Roland H.C. Yap. An extended Low Fat allocator API and applications. *arXiv preprint arXiv:1804.04812*, 2018.

[65] Gregory J. Duck and Roland H.C. Yap. Heap bounds protection with Low Fat Pointers. In *CC*, 2016.

[66] Gregory J. Duck, Roland H.C. Yap, and Lorenzo Cavallaro. Stack bounds protection with Low Fat Pointers. In *NDSS*, 2017.

[67] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of Heartbleed. In *IMC*, 2014.

[68] Jake Edge. Kernel address space layout randomization. `https://lwn.net/Articles/569635/`, 2013. Accessed: 2018-02-05.

[69] Robert van Engelen. Efficient symbolic analysis for optimizing compilers. In *CC*, 2001.

[70] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: software guards for system address spaces. In *OSDI*, 2006.

[71] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *S&P*, 2015.

[72] Edward A. Feustel. On the advantages of tagged architecture. *IEEE Transactions on Computers*, 1973.

[73] Agner Fog. Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering*, 2016.

[74] Bryan Ford and Russ Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX ATC*, 2008.

[75] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse computer systems. In *HotOS*, 1997.

[76] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *NDSS*, 2016.

[77] Robert Gawlik, Philipp Koppe, Benjamin Kollenda, Andre Pawlowski, Behrad Garmany, and Thorsten Holz. Detile: fine-grained information leak detection in script engines. In *DIMVA*, 2016.

[78] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security*, 2012.

[79] Enes Gökta, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: overcoming control-flow integrity. In *S&P*, 2014.

[80] Enes Gökta, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security*, 2014.

[81] Enes Gökta, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining information hiding (and what to do about it). In *USENIX Security*, 2016.

[82] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: practical cache attacks on the MMU. In *NDSS*, 2017.

[83] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *S&P*, 2015.

[84] David Gudeman. *Representing type information in dynamically typed languages*. 1995.

[85] Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *OOPSLA*, 2009.

[86] Dave Hansen. Linux: x86 MPX removal. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ccaaaf6fe5a5e1fffca5cca0f3fc4ec84d7ae752`, 2020.

[87] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *CGO*, 2012.

[88] John L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 2006.

[89] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: where'd my gadgets go? In *S&P*, 2012.

[90] Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *ICSE*, 2013.

[91] Petr Hosek and Cristian Cadar. VARAN the unbelievable: an efficient N-version execution framework. In *ASPLOS*, 2015.

[92] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *USENIX Security*, 2015.

[93] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *S&P*, 2013.

[94] Intel Corporation. Control-flow Enforcement Technology Specification. Technical report, May 2019.

[95] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. January 2016.

[96] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. June 2017.

[97] ISO/IEC. International Standard ISO/IEC 9899:1999 - Programming Languages - C, 1999.

[98] Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation – a Pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites. *VSSAD Technical Report*, 2007.

[99] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of C. In *USENIX ATC*, 2002.

[100] Richard W.M. Jones and Paul H.J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG*, 1997.

[101] Mark K. Joseph and Algirdas Avižienis. A fault tolerance approach to computer viruses. In *S&P*, 1988.

[102] David Kaplan, Jeremy Powell, and Tomm Woller. AMD Memory Encryption. Technical report, AMD, April 2016. URL: `http://bit.ly/2gr5hQM`.

[103] Kaspersky Lab. The financial impact of IT security on US businesses, 2016.

[104] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. Ret2dir: rethinking kernel isolation. In *USENIX Security*, 2014.

[105] Dohyeong Kim, Yonghwi Kwon, William N. Sumner, Xiangyu Zhang, and Dongyan Xu. Dual execution for on the fly fine grained execution comparison. In *ASPLOS*, 2015.

[106] Tobias Klein. CVE-2009-0385, 2009.

[107] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *DSN*, 2016.

[108] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: protecting safe regions on commodity hardware. In *EuroSys*, 2017.

[109] Greg Kroah-Hartman. The Linux Kernel Driver Interface. https://www.kernel.org/doc/Documentation/process/stable-api-nonsense.rst, 2018. Accessed: 2018-01-22.

[110] Taddeus Kroes, Koen Koning, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Fast and generic metadata management with mid-fat pointers. In *EuroSec*, 2017.

[111] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A Linux in unikernel clothing. In *EuroSys*, 2020.

[112] Anil Kurmus and Robby Zippel. A tale of two kernels: towards ending kernel hardening wars with split kernel. In *CCS*, 2014.

[113] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. SGXBounds: memory safety for shielded execution. In *EuroSys*, 2017.

[114] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *OSDI*, 2014.

[115] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. LDX: causality inference by lightweight dual execution. In *ASPLOS*, 2016.

[116] Michael Larabel. Intel MPX Support Removed From GCC 9. https://www.phoronix.com/scan.php?page=news_item&px=MPX-Removed-From-GCC9, 2018.

[117] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: automated software diversity. In *S&P*, 2014.

[118] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[119] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[120] Linux Kernel. Coccinelle. https://static.lwn.net/kerneldoc/dev-tools/coccinelle.html, 2018. Accessed: 2018-01-22.

[121] Linux Kernel. Getting started with kmemcheck. https://www.kernel.org/doc/Documentation/dev-tools/kmemcheck.rst, 2007. Accessed: 2018-01-22.

[122] Linux Kernel. Sparse. https://static.lwn.net/kerneldoc/dev-tools/sparse.html, 2018. Accessed: 2018-01-22.

[123] Linux Kernel. The Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/Documentation/dev-tools/kasan.rst, 2015. Accessed: 2018-01-22.

[124] David Litchfield. Buffer Underruns, DEP, ASLR and improving the Exploitation Prevention Mechanisms (XPMs) on the Windows platform. Technical report, NGSSoftware Insight Security Research (NISR), 2005.

[125] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In CCS, 2018.

[126] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *CCS*, 2015.

[127] Giuliano Losa, Antonio Barbalace, Yuzhong Wen, Marina Sadini, Ho-Ren Chuang, and Binoy Ravindran. Transparent fault-tolerance using intra-machine full-software-stack replication on commodity multicore hardware. In *ICDCS*, 2017.

[128] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. UniSan: proactive kernel memory initialization to eliminate data leakages. In *CCS*, 2016.

[129] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: stopping address space leakage for code reuse attacks. In *CCS*, 2015.

[130] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nürnberger, Wenke Lee, and Michael Backes. Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying. In *NDSS*, 2017.

[131] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[132] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: cryptographically enforced control flow integrity. In *CCS*, 2015.

[133] Matthew Maurer and David Brumley. Tachyon: tandem execution for efficient live patch testing. In *USENIX Security*, 2012.

[134] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *USENIX Security*, 2006.

[135] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: efficient TCB reduction and attestation. In *S&P*, 2010.

[136] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for TCB minimization. In *EuroSys*, 2008.

[137] Microsoft. A detailed description of the Data Execution Prevention (DEP) feature. `http://support.microsoft.com/kb/875352`, 2006.

[138] Microsoft. Microsoft Control Flow Guard. `https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard`, 2018.

[139] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. SafeInit: comprehensive and practical mitigation of uninitialized read vulnerabilities. In *NDSS*, 2017.

[140] Matt Miller. Heap corruption issues reported to Microsoft. https://twitter.com/epakskape/status/851479629873332224, 2017.

[141] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf, February 2019.

[142] MITRE Corporation. 2019 CWE Top 25 Most Dangerous Software Errors. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.

[143] MITRE Corporation. Common vulnerabilities and exposures. http://cve.mitre.org.

[144] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. Opaque control-flow integrity. In *NDSS*, 2015.

[145] Ingo Molnar. fs/proc, core/debug: Don't expose absolute kernel addresses via wchan. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b2f73922d119686323f14fbbe46587f863852328, 2015. Accessed: 2018-01-30.

[146] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. 1997.

[147] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: a distributed operating system for the 1990s. *IEEE Computer*, 23(5), May 1990.

[148] Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Watchdog: hardware for safe and secure manual memory management and full memory safety. In *ISCA*, 2012.

[149] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: compiler-enforced temporal safety for C. In *ISMM*, 2010.

[150] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *PLDI*, 2009.

[151] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *POPL*, 2002.

[152] Ruslan Nikolaev and Godmar Back. VirtuOS: an operating system with kernel virtualization. In *SOSP*, 2013.

[153] Ben Niu and Gang Tan. Monitor integrity protection with space efficiency and separate compilation. In *CCS*, 2013.

[154] Ben Niu and Gang Tan. Per-input control-flow integrity. In *CCS*, 2015.

[155] Gene Novark and Emery D. Berger. DieHarder: securing the heap. In *CCS*, 2010.

[156] Angelos Oikonomopoulos, Cristiano Giuffrida, Elias Athanasopoulos, and Herbert Bos. Poking holes into information hiding. In *USENIX Security*, 2016.

[157] Peter Okech, Nicholas Mc Guire, Christof Fetzer, and William Okelo-Odongo. Investigating execution path non-determinism in the Linux kernel. In *OSADL*, 2013.

[158] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: a cross-layer analysis of the Intel MPX system stack. *SIGMETRICS*, 2018.

[159] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. *ASPLOS*, 2009.

[160] Oracle. M7: next generation SPARC, 2014.

[161] Serkan Özkan. CVE Details Linux Kernel. `https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33`, 2017. Accessed: 2018-01-22.

[162] Serkan Özkan. Linux Kernel : Security Vulnerabilities Published In 2017 (Gain Information). `https://www.cvedetails.com/vulnerability-list.php?vendor_id=33&product_id=47&opginf=1&year=2017`, 2017. Accessed: 2018-04-30.

[163] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. Digtool: a virtualization-based framework for detecting kernel vulnerabilities. In *USENIX Security*, 2017.

[164] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: hindering return-oriented programming using in-place code randomization. In *S&P*, 2012.

[165] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security*, 2013.

[166] T. Paul Parker and Shouhuai Xu. A method for safekeeping cryptographic keys from memory disclosure attacks. In *INTRUST*, 2009.

[167] PaX Team. Grsecurity and PaX Configuration Options. `https://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options`, 2017. Accessed: 2018-01-22.

[168] PaX Team. PaX - gcc plugins galore. `https://pax.grsecurity.net/docs/PaXTeam-H2HC13-PaX-gcc-plugins.pdf`, 2018. Accessed: 2018-01-22.

[169] PaX Team. PaX Address Space Layout Randomization. `https://pax.grsecurity.net/docs/aslr.txt`.

[170] Mathias Payer, Antonio Barresi, and Thomas R. Gross. Fine-grained control-flow integrity through binary hardening. In *DIMVA*, 2015.

[171] S. Peiró, M. Muñoz, M. Masmano, and A. Crespo. Detecting stack based kernel information leaks. In *CISIS*, 2014.

[172] Simon Peter, Jialin Li, Irene Zhang, Dan R.K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: the operating system is the control plane. In *OSDI*, 2014.

[173] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. MVEDSUA: higher availability dynamic software updates via multi-version execution. In *ASPLOS*, 2019.

[174] Luís Pina, Daniel Grumberg, Anastasios Andronidis, and Cristian Cadar. A DSL approach to reconcile equivalent divergent program executions. In *USENIX ATC*, 2017.

[175] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. kR^X: comprehensive kernel protection against just-in-time code reuse. In *EuroSys*, 2017.

[176] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid Android: versatile protection for smartphones. In *ACSAC*, 2010.

[177] Niels Provos. Improving host security with system call policies. In *USENIX Security*, 2003.

[178] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.

[179] Babak Salamat. *Multi-Variant Execution: Run-Time Defense against Malicious Code Injection Attacks DISSERTATION*. PhD thesis, University of California, Irvine, 2009.

[180] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *EuroSys*, 2009.

[181] Charles Schmidt and Tom Darby. The What, Why, and How of the 1988 Internet Worm. `https://ethics.csc.ncsu.edu/abuse/wvt/worm/darby/worm.html`.

[182] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: hardware-assisted feedback fuzzing for OS kernels. In *USENIX Security*, 2017.

[183] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: on the difficulty of preventing code reuse attacks in C++ applications. In *S&P*, 2015.

[184] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *USENIX Security*, 2010.

[185] Jeff Seibert, Hamed Okhravi, and Eric Söderström. Information leaks without memory disclosures: remote side channel attacks on diversified code. In *CCS*, 2014.

[186] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: a fast address sanity checker. In *USENIX ATC*, 2012.

[187] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves C/C++ memory safety. *arXiv preprint arXiv:1802.09517*, 2018.

[188] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS*, 2007.

[189] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM monitoring using hardware virtualization. In *CCS*, 2009.

[190] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time code reuse: on the effectiveness of fine-grained address space layout randomization. In *S&P*, 2013.

[191] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *OSDI*, 2010.

[192] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[193] Dannie M. Stanley, Dongyan Xu, and Eugene H. Spafford. Improved kernel security through memory layout randomization. In *IPCCC*, 2013.

[194] stress-ng team. stress-ng: a tool to load and stress a computer system. `http://kernel.ubuntu.com/~cking/stress-ng/`, 2018. Accessed: 2018-04-30.

[195] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: eternal war in memory. In *S&P*, 2013.

[196] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: thwarting memory disclosure attacks using destructive code reads. In *CCS*, 2015.

[197] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. Disruptor: high performance alternative to bounded queues for exchanging data between concurrent threads. *Technical paper. LMAX*, 2011.

[198] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security*, 2014.

[199] Ubuntu. Security Features. `https://wiki.ubuntu.com/Security/Features`.

[200] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: secure, efficient in-process isolation with protection keys (MPK). In *USENIX Security*, 2019.

[201] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. Type-After-Type: practical and complete type-safe memory reuse. In *ACSAC*, 2018.

[202] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: scalable use-after-free detection. In *EuroSys*, 2017.

[203] Victor van der Veen, Dennis Andriesse, Enes Gökta, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *CCS*, 2015.

[204] Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: the past, the present, and the future. In *RAID*, 2012.

[205] Victor van der Veen, Enes Gökta, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: mitigating advanced code-reuse attacks at the binary level. In *S&P*, 2016.

[206] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: protecting software with code-centric memory domains. In *ISCA*, 2014.

[207] Stijn Volckaert, Bart Coppens, and Bjorn de Sutter. Cloning your gadgets: complete ROP attack immunity with multi-variant execution. *IEEE TDSC*, 13(4), 2016.

[208] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn de Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In *USENIX ATC*, 2016.

[209] Stijn Volckaert, Bjorn de Sutter, Tim de Baets, and Koen de Bosschere. GHUMVEE: efficient, effective, and flexible replication. In *FPS*, 2012.

[210] Vrije Universiteit Amsterdam. The distributed ASCI supercomputer 5. `https://www.cs.vu.nl/das5`.

[211] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[212] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary Stirring: self-randomizing instruction addresses of legacy x86 binary code. In *CCS*, 2012.

[213] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: fast and deployable continuous code re-randomization. In *OSDI*, 2016.

[214] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: revisiting RISC in an age of risk. In *ISCA*, 2014.

[215] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N.M. Watson, and Timothy M. Jones. CHERIvoke: characterising pointer revocation using CHERI capabilities for temporal memory safety. In *MICRO*, 2019.

[216] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. *FSE*, 2004.

[217] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: unleashing use-after-free vulnerabilities in Linux kernel. In *CCS*, 2015.

[218] Jisoo Yang and Kang G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *VEE*, 2008.

[219] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: a sandbox for portable, untrusted x86 native code. In *S&P*, 2009.

[220] Yves Younan. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.

[221] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. PAriCheck: an efficient pointer arithmetic checker for C programs. In *ASIACCS*, 2010.

[222] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *S&P*, 2013.

[223] Fengwei Zhang and Hongwei Zhang. SoK: a study of using hardware-assisted isolated execution environments for security. In *HASP*, 2016.

[224] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security*, 2013.

[225] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: buy spatial memory safety, get temporal memory safety (almost) free. In *ASPLOS*, 2019.

[226] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: hardware-based fault isolation for ARM. In *CCS*, 2014.

Summary

Computer software is riddled with bugs caused by programmer errors. While many of such bugs are benign, attackers may leverage some of these bugs to create exploits with the purpose of stealing or modifying information. A significant portion of the bugs used by attackers are so-called *memory errors*. These errors exist in low-level programming languages, where the programmer is responsible for the error-prone process of manually managing the memory of the computer. While modern, high-level, programming languages do not suffer from such memory errors, unsafe languages are still ubiquitous. For example, operating systems, web browsers and servers are often written in the unsafe C and C++ languages. For over three decades, researchers have sought out solutions to automatically and transparently protect unsafe software. Hardening software with checks against memory errors is a difficult process though, for such defenses often suffer from issues with performance, compatibility and completeness.

This dissertation provides new insights and designs for memory safety solutions in terms of performance, compatibility and/or security. We focus on the usage of common, widely available, hardware features and extensions to improve existing solutions. Hardware can often be used to do operations more efficiently, but developing custom hardware is costly and time-consuming. Leveraging existing hardware, such as the virtualization extensions of a processor, allows for more efficient and transparent security checks, while still being widely deployable on existing machines.

Using a variety of different processor features we can increase the security of many available software defenses in an efficient way. Many existing defenses maintain metadata which is integral to the security of the defense, but fail to properly protect this information. We consider a number of different extensions, including Intel VMX, MPX and MPK, and analyze how these can be used to securely protect sensitive data against attackers.

With Delta Pointers we present a new design for a bounds checker, a system which protects against spatial memory errors. Our design omits both separate metadata and explicit checks, by encoding all required information inside pointers themselves. Through special encoding, our design is able to delegate checks to

existing processor hardware, optimizing checks significantly.

For protecting binaries where source code modifications and compiler transformations are not possible, we present MvArmor. We build on top of the multi-variant execution (MVX) design, where multiple diversified variants of the same binary are run simultaneously. By observing differences between the execution of these variants, security exploits can be detected. MvArmor presents new variation techniques for increased security, and moreover, is able to efficiently monitor the execution through light-weight virtualization.

Finally, we present a new MVX design tailored towards protecting operating system kernels. kMVX tackles the challenges of how to run multiple kernels simultaneously, and where and how to synchronize them. We show new variation techniques suitable for mitigating information leaks in the kernel.

Samenvatting

Computersoftware zit vol bugs die worden veroorzaakt door programmeerfouten. Hoewel de meeste bugs niet schadelijk zijn, kunnen sommige door aanvallers misbruikt worden om data te stelen of informatie aan te passen.

Een groot deel van deze bugs zijn zogenaamde geheugenfouten. Deze fouten komen voor in *low-level* programmeertalen, waar programmeurs zelf verantwoordelijk zijn voor het beheren van het geheugen van de computer. Hoewel moderne, *high-level*, programmeertalen geen last hebben van dergelijke problemen, worden "onveilige" talen nog veelvuldig gebruikt. Onder andere besturingssystemen, browsers en servers zijn nog vaak geschreven in de "onveilige" talen C en C++.

Onderzoekers zoeken al drie decennia naar oplossingen om software automatisch en transparant te beschermen. Beveiliging van software tegen geheugenfouten is helaas een lastig probleem, en veel oplossingen schieten tekort op het gebied van efficiëntie, compatibiliteit en volledigheid.

Dit proefschrift biedt nieuwe inzichten en ontwerpen op het gebied van geheugenbeveiliging. Wij focussen op het gebruik van bestaande, algemeen verkrijgbare hardware. Hardware kan vaak bepaalde operaties veel sneller doen dan software, maar kost veel tijd en geld om te ontwikkelen. Door reeds bestaande hardware te gebruiken kunnen we efficiëntere beveiliging geven die ook werkt op bestaande computers.

Door een verscheidenheid van processor-uitbreidingen kunnen wij de veiligheid van bestaande beveiligingsoplossingen verbeteren. Veel bestaande oplossingen gebruiken metadata die cruciaal is voor de veiligheid, maar beschermen deze metadata zelf vaak niet. Wij bekijken een aantal verschillende processor-uitbreidingen, waaronder Intel VMX, MPX, en MPK, en analyseren hoe we deze kunnen gebruiken om gevoelige data te beschermen.

Met Delta Pointers presenteren wij een nieuw ontwerp voor een *bounds checker*, een systeem dat verdedigt tegen *buffer overflows*. Ons ontwerp vereist geen extra metadata en expliciete controles, doordat het alle benodigde informatie direct in pointers zelf plaatst. Door speciale codering kan de hardware zelf controles uitvoeren, wat het ontwerp zeer efficiënt maakt.

Om reeds gecompileerde programma's te beschermen, waar broncode-

aanpassingen en compiler-transformaties niet mogelijk zijn, presenteren wij MvArmor. Hier bouwen wij voort op het *multi-variant execution* (MVX) idee, waar meerdere gediversifieerde varianten van hetzelfde programma tegelijk draaien. Door verschillen tussen het gedrag van deze varianten te observeren kunnen we aanvallen detecteren en stoppen. MvArmor presenteert nieuwe variatietechnieken voor hogere veiligheid, en gebruikt virtualisatie om deze varianten efficiënt te controleren.

Tot slot presenteren wij een nieuw MVX ontwerp dat kernels van besturingssystemen kan beschermen. kMVX presenteert oplossingen voor het draaien van meerdere kernels op dezelfde computer en hoe we deze kunnen synchroniseren. Onze nieuwe variatietechnieken beschermen de kernel tegen het lekken van gevoelige informatie.